ISSN 0265-2919

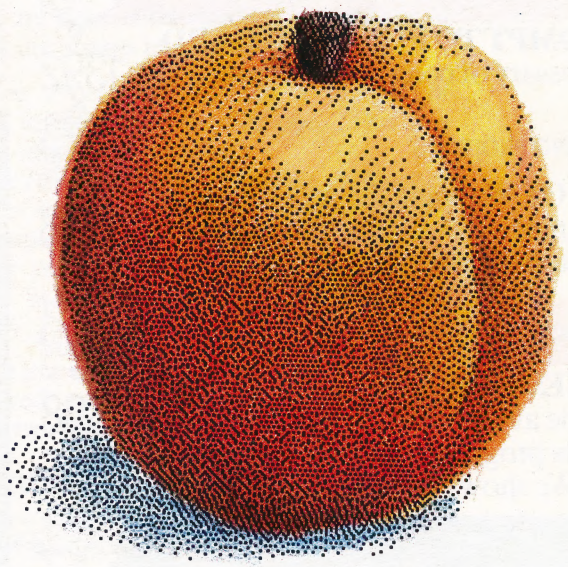
90p

70

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

Aa



File

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

EYE UNDERSTAND An AI system must do more than simply perceive patterns in the environment around it. We look at some of the problems involved in programming computers to make sense of what they see — 1381

HARDWARE

A LITTLE PEACH A taste of things to come! The ACT Apricot F1e is one of the first entries in the impending marketting race to provide high-powered and affordable machines for the serious home micro user — 1389

SOFTWARE

PLANNING THE ACTION We take a look at two of the best program generators — TLO and Sycero — in action — 1396

INTERSTELLAR INTERACTION Now you can join Ford Prefect, Zaphod Beeblebrox and Arthur Dent in a text-only adventure of a lifetime — 1400

COMPUTER SCIENCE

FUNCTIONAL DESIGN This week we look at several of LISP's more important functions — 1384

JARGON

FROM PROMPT TO PUNCHED CARD A weekly glossary of computing terms — 1392

PROGRAMMING PROJECTS

SET-UP SHOT We begin the programming of our Go game by developing procedures to set up the board display — 1393

MACHINE CODE

HOOK LINES There are a variety of useful routines made available to the Spectrum machine code programmer using the Interface 1. We show you how to use them — 1398

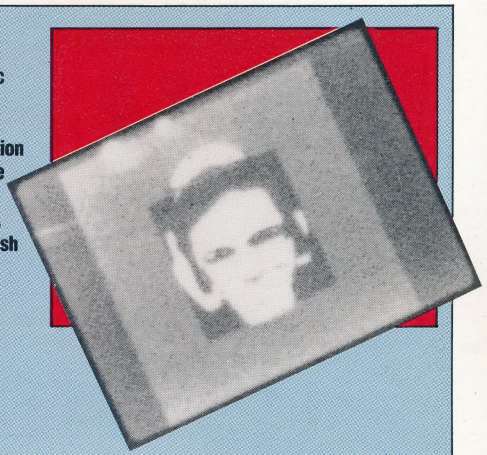
WORKSHOP

RULES OF COMPOSITION We look into some aspects of data transmission between a computer and an instrument — 1386

REFERENCE CARD Another section of the Commodore 64's memory map — INSIDE BACK COVER

Next Week

- **We look inside the Magic Mouse system for the Commodore 64.**
- **The Go game's initialisation listings are provided for the Amstrad, Spectrum and Commodore 64 computers.**
- **In our AI series, we publish a pattern recognition program.**



QUIZ

1) In the language LISP, what is the difference between a list and a function?
2) How does an instrument register that a new system message is arriving using a MIDI system?
3) What is a 'controlled hallucination'?
4) What is a 'hook code'?

Answers To Last Week's Quiz

1) While text can be input in a standard form, variations in speech and pronunciation mean that the computer has to be able to deal with wide variations in the possible inputs.

2) The name LISP is derived from 'LISt Processing'. The language has generated interest in the field of AI because of the way in which it can search lists of data for possible solutions to problems.

3) An interrupt vector is the address that the processor will jump to when it encounters an interrupt.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Deputy Editor** Steve Colwill; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistants** Caroline Clayton, Mike Clowes; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Karl Dallas, Martin Young, Joe Pritchard, Steve Malone, Tony Samuels, Steve Colwill, Steve Cooke; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

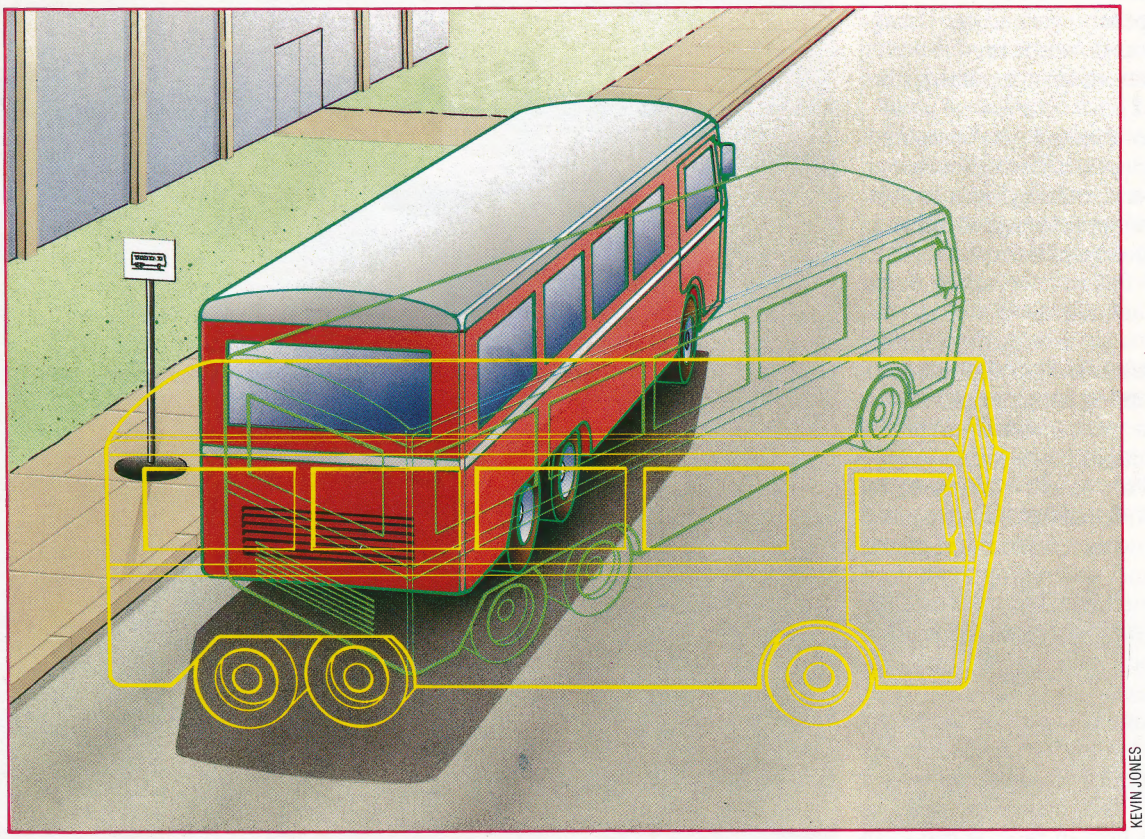
HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. **ADDRESS FOR BINDERS AND BACK ISSUES** – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.

NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER GRAPHICS BY IAN McKINNELL ON THE MACINTOSH

EYE UNDERSTAND



KEVIN JONES

A Good Fit
Some pattern recognition systems adopt a 'top-down' approach in which a pattern or scene is scanned for the presence of a particular object. A wire-framed representation of the object is projected onto the screen in various orientations until, if the object is present, a projection is found that fits the outline of the actual figure. In a three-dimensional scene, such as that shown here, there are a large number of possible projections of a wire-framed model of the bus that could be made before the correct one was found

Because seeing and understanding are such integral (and interrelated) factors in the way we perceive our environment, we often find it difficult to separate the two, which is exactly what is needed when programming a computer to 'understand' what it 'sees'.

'Beauty is in the eye of the beholder', as they say; but it should be said that beauty is in the brain of the perceiver – or, more accurately, in a complex chain of neural processes that begins at the back of the retina and ends somewhere in the occipital cortex of the brain. It is this chain of events that physiologists seek to understand and roboticists, in some measure, seek to duplicate (for examples of the latter see pages 681 and 741).

Visual perception is such an integral component of the way in which we understand our environment that we often say 'I see' when we mean 'I understand'. Indeed, understanding is the key to the challenge of computer vision. Somehow the computer must take information supplied by a camera, or other light-sensitive device, and realise what that information tells it about the state of its environment. Acquiring the information is easy — the problem is how to interpret it.

There are three main stages in the process of transforming images into meanings:

1. *Picture Processing:* Turning a blurred or distorted image into a sharper one. (This is a relatively simple task to solve.)
2. *Pattern Recognition:* Detecting the presence or absence of significant features or objects. (This is more complex.)
3. *Image understanding:* Working out what is going on in the real world. (A very complex problem.)

Although true computer vision (Stage 3) has yet to be achieved, useful results have been obtained from the two initial stages.

PATTERN RECOGNITION

Pattern recognition is a challenging task, though we find it hard to 'see' the difficulties involved. Pattern recognisers classify images by matching them with a limited set of alternatives, such as the letters of the alphabet for optical character recognition systems. Typically, they extract features by examining parts of the digitised image (small groups of pixels). The system can then recognise and classify patterns based on the presence or absence of certain distinctive features.

The recognition of some features, such as diagonal strokes, seems clear to us, but often recognition by such systems can result from purely arbitrary computations performed on the image data. Such systems have important practical uses, but they hardly scratch the surface of what we mean by vision.

Historically, there have been two contrasting approaches to computer vision as image understanding — the top-down or model-driven approach and the bottom-up or data-driven approach (see page 742).

Model-driven scene analysis is sometimes called 'controlled hallucination'.The system has an internal model of what it is looking for (a double-decker bus, let's say) and it projects that model in various orientations onto the picture plane. Then it tests for quality of fit between what it has 'imagined' and what is actually there.

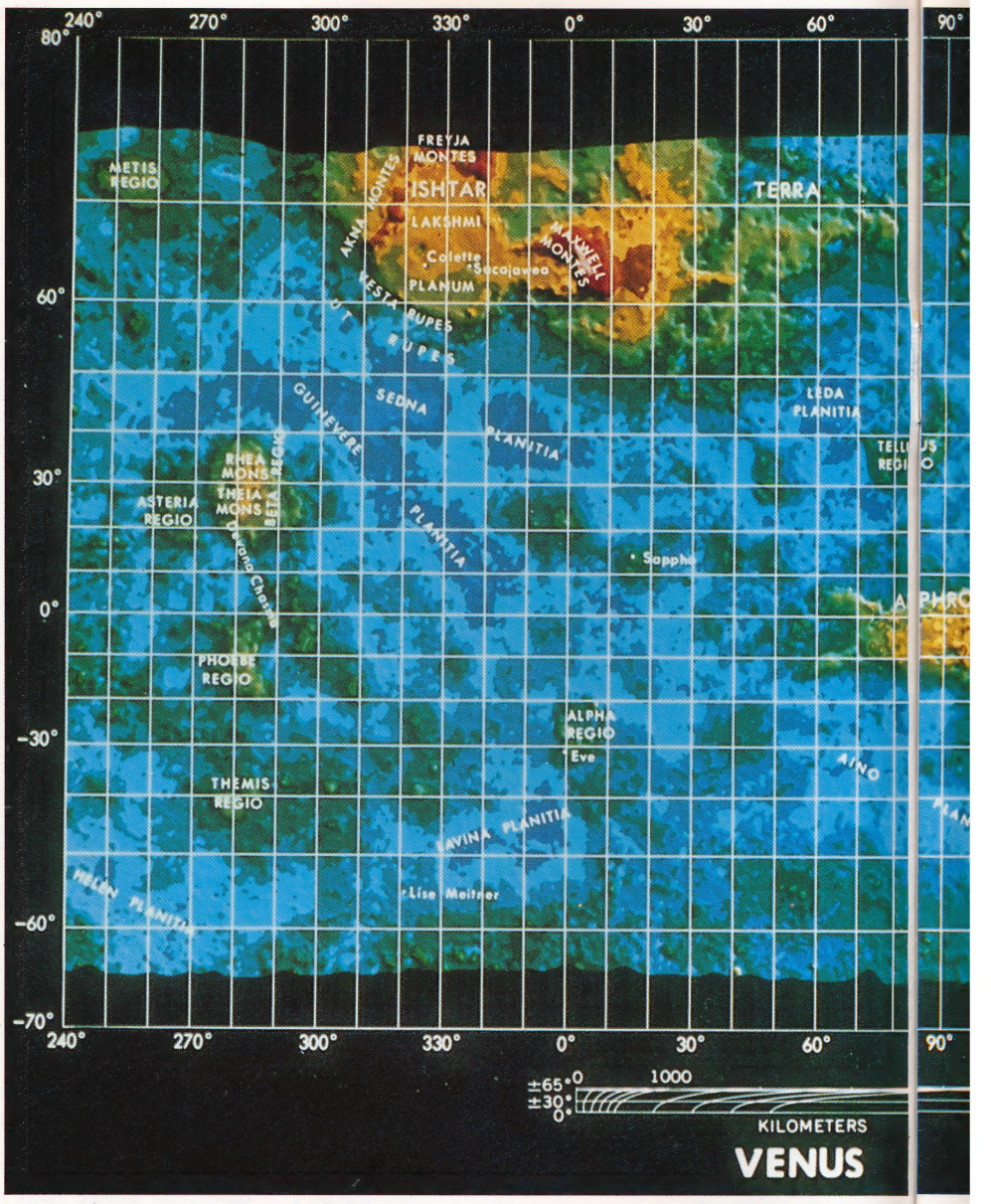
Bottom-up systems scan the image data for lines, edges and other tell-tale signs. By doing so, they construct a simplified description of the image, which is termed a 'primal sketch' — a highly abstracted data representation. The idea is that by stripping away much of the detail, it will be easier to compare the sketch with examples from a given category of objects stored as reference templates.

Practical systems tend to employ both top-down and bottom-up methods, but until recently there were very few learning systems in the field of computer vision. They all relied on pre-programmed intelligence.

A notable exception is Igor Aleksander's WISARD (Wilkie, Aleksander & Stonham's Recognition Device). It can be taught to make the comparatively subtle distinction between a smiling and a frowning human face. It can even generalise this knowledge to faces it has never seen before. WISARD operates in real-time (25 frames per second), and its commercial success in grading chocolates on a moving conveyor belt may pave the way for a new generation of 'seeing machines' that adapt and improve their performances.

Very briefly, WISARD works by assigning groups of eight pixels at a time (octuples) to selected banks of RAM. An octuple is a feature detector, and it can be in one of 256 states (0 to 255) depending on the status of each of the pixels it monitors. During the training phase, a 1 is stored at the location within its RAM-bank specified by the state of the octuple when a particular image is present. Later, in the recognition phase, the presence of a 1 at the addressed location within that octuple's RAM-bank is evidence that the 'taught' image is again present.

By using a large number of octuples, or discriminators, the system becomes relatively impervious to spurious data ('noisy' images). Aleksander's system has a 512 by 512 grid for the image and over 32,000 octuples. This requires about eight million bits (a megabyte) of RAM. Only recently have such memories become economically feasible.



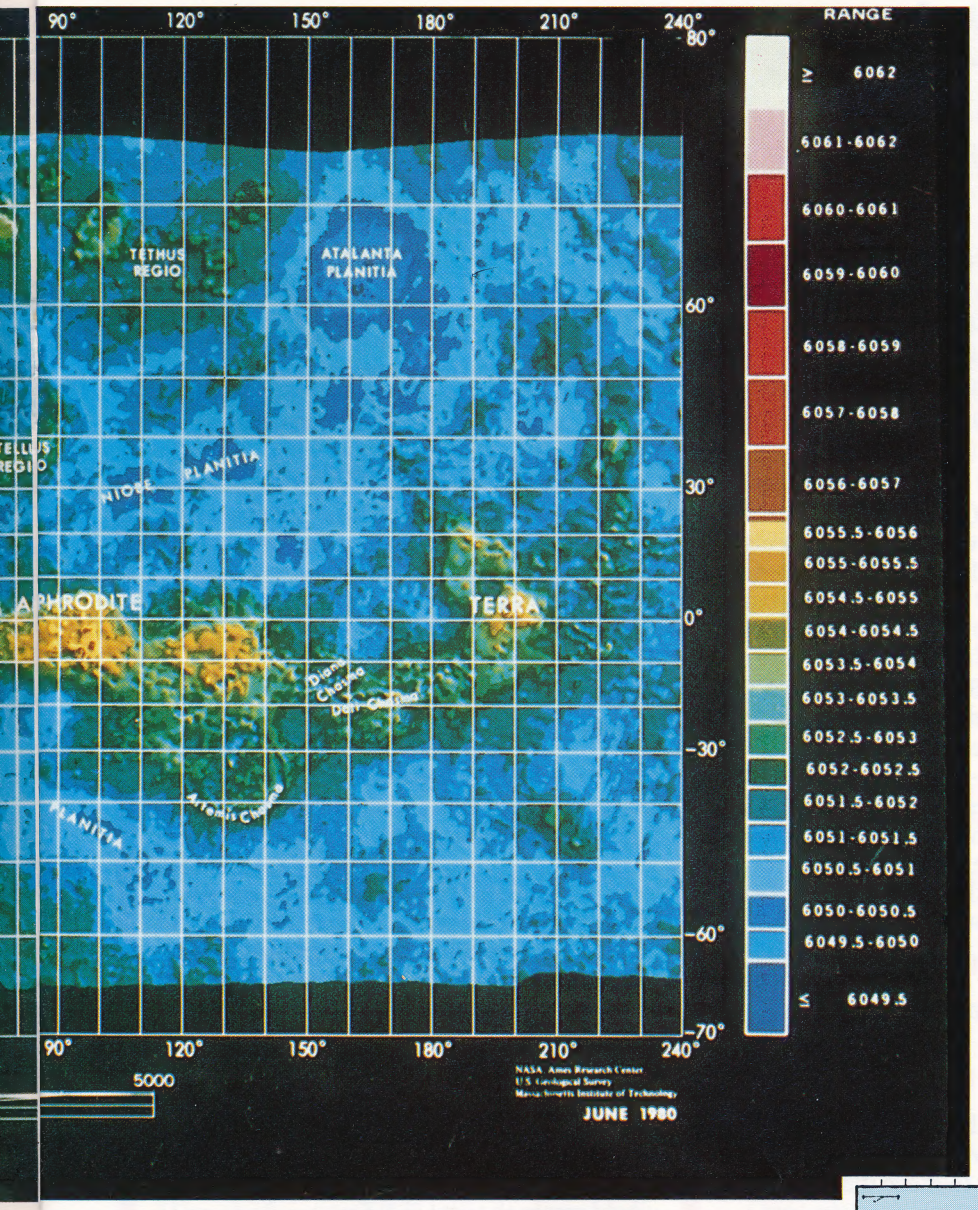
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	1	1	0	0
0	1	1	0	0	0	1	0
0	0	1	1	1	1	0	0
0	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	0	0	0	1	0

0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0
0	0	1	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	1	1	1	0	0	0
0	1	1	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	0	1	0

CAROLINE CLAYTON

The Difference Between R And A

In the real world well-defined data seldom exists. The above example shows digitised forms of the letters A and R, but neither pattern is well-defined, having smudges and distortions. 'Noisy' data like this is a perennial problem for all types of AI systems. Can you tell which is which?



Picture Processing

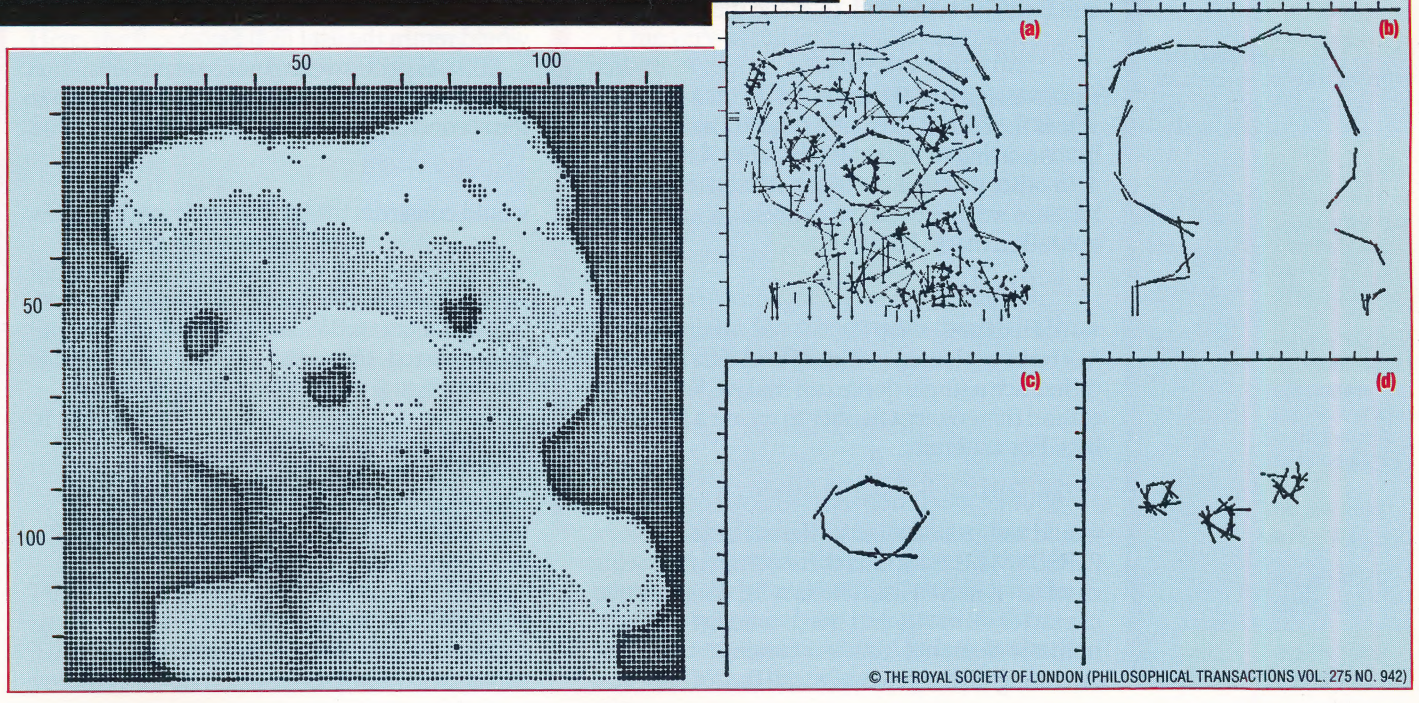
An example of picture processing is illustrated. Here, a picture taken from a space probe has been processed by computer to accentuate the boundaries between regions of different surface elevations. This involves the use of 'false colours'.

The interesting thing about this picture, which is in fact a radar image of the planet Venus, is that all the colours are 'false'. We could never see this image without the help of the computer because Venus is swathed in thick cloud, and our eyes are not sensitive to the wavelengths required to penetrate it. In astronomy, such pictures have proved very useful (and sometimes very beautiful) over recent years, taking advantage of radio, infra-red and ultraviolet wavelengths as well as visible light. Closer to home, the ocean bed is being surveyed by sonar to build up 'acoustic pictures' that use the echo-sounding principle and do not rely on electromagnetic radiation (visible light, ultraviolet, infra-red, x-rays and so on) at all.

Other kinds of picture processing are used to 'clean up' smudged or distorted images. Such algorithms are now used so routinely that they no longer qualify as being components of AI

Primal Sketches

A method of visual processing has been developed that does not rely on prior knowledge of what the pattern is meant to represent. The primal sketch (a) extracts boundaries and primitive shapes from the image of the toy bear by comparing adjacent regions. Further extractions from the primal sketch — (b) to (d) — show important groups that assist recognition of the original image



© THE ROYAL SOCIETY OF LONDON (PHILOSOPHICAL TRANSACTIONS VOL. 275 NO. 942)

FUNCTIONAL DESIGN

As we continue our series on LISP, we examine the two fundamental concepts of the language — lists and functions. We show how it is possible to create lists within lists, and discuss the importance of conditional statements and defining our own functions.

In the first instalment of this series, we saw how to inform LISP that we wanted a particular list to be evaluated as a list of data items, rather than as a function. So:

```
(SETQ X '(A B C D E F) )
```

would assign the list (A B C D E F) to the variable X, where each element of the list is a single-character string. Let's see what would happen if D, E and F were single-character strings and A, B and C were variables with the values 2, 4 and 8, respectively. In this case, we would really want to assign the list (2 4 8 D E F) to X. We can do this by introducing a new function: LIST. This creates a list of its arguments. Thus:

```
(SETQ X (LIST 'A 'B 'C 'D 'E 'F) )
```

would be exactly the same as the previous expression, but:

```
(SETQ X LIST A B C 'D 'E 'F) )
```

would correctly assign the list (2 4 8 D E F). Here, by omitting the preceding quotes, A, B and C will be evaluated. In much the same way:

```
(SETQ X (LIST 1 2 4 '(PLUS 4 4) )
```

would assign to X the list (1 2 4 (PLUS 4 4)) where the fourth element of the list is itself a list consisting of the three elements PLUS, 4 and 4. In other words, PLUS has not been evaluated because we preceded the expression with a quotation mark. However:

```
(SETQ X (LIST 1 2 4 (PLUS 4 4) )
```

would assign to X the list (1 2 4 8), where PLUS has been evaluated (which is evidence of the difference a single quote can make). We can extend this concept further to create a list of lists. For example:

```
(SETQ PERSON '( (JOE BLOGGS) (17 1 1960) ) )
```

would assign a data list to the variable PERSON. This list contains two elements, each of which is itself a list. The first list contains two character elements and the second list three numeric elements. You can continue extending this to get even more lists of lists.

So far, we have looked at the functions SETQ, LIST, PLUS and TIMES. There are many more functions built into LISP, and how they are used will depend on the particular implementation. In addition, you may define your own functions — in fact, this is the way in which a LISP program is built.

Before looking at how this is done, let's examine the three basic functions found in all implementations of LISP. These are CAR, CDR and CONS. The name CONS simply stands for CONStruct. The names CAR and CDR date from one of the original implementations of the language, and are acronyms for Contents of Address Register and Contents of Decrement Register, respectively.

CAR and CDR both take a single argument, which must be a list with at least one element. This excludes the special list structure known as the 'empty list', or NIL, which is written:

```
( )
```

The function CAR returns as its result the first element of its list argument. Thus:

```
(CAR '(1 2 3 4 5) )
```

would return the integer value 1. There is no reason why the result should not itself be a list. For example, in the expression:

```
(CAR '( (1 2) (3 4) 5) )
```

the first element is the list (1 2).

The function CDR effectively returns everything but the first element of a list. In other words, it returns what is left after a CAR operation. So:

```
(CDR '(1 2 3 4 5) )
```

would return the list (2 3 4 5), and:

```
(CDR '( (1 2) (3 4) 5) )
```

would return the list ((3 4) 5).

CONS expects two arguments and will concatenate them, adding the first argument to the second argument list. Thus:

```
(CONS 1'  (2 3 4 5) )
```

would construct the list (1 2 3 4 5), and:

```
(CONS '(1 2) '( (3 4) 5) )
```

would construct the list ((1 2) (3 4) 5).

The functions CAR and CDR will often be found nested, so:

```
(CDR '( (1 2) (3 4) 5) )
```

will return ((3 4) 5), and:

```
(CAR (CDR '( (1 2) (3 4) 5) ) )
```

will return (3 4) which is (CAR '((3 4) 5)), and:

```
(CAR (CAR (CDR '( (1 2) (3 4) 5) ) ) )
```

will return the value 3, which is (CAR '(3 4)).

This can soon become tedious, so LISP usually allows abbreviations. The function

names still start with a C and end with R but may have any combination of As (for CAR) and Ds (for CDR), in between. Using Acornsoft LISP, this combination may consist of up to three embedded letters. Thus we could write the last expression as:

```
(CAAR (CDR '( (1 2) (3 4) 5) ) )
```

or:

```
(CAR (CADR '( (1 2) (3 4) 5) ) )
```

or even just:

```
(CAADR '( (1 2) (3 4) 5) )
```

all of which return the answer 3.

DEFINING FUNCTIONS

As we've noted already, LISP programs are built up as a series of functions that are defined by the user. We have seen that all LISP expressions are function lists, so quite naturally we use functions to define functions. In this case, we use the function DEFUN,which expects three arguments of the form:

```
(DEFUN a (b) (c) )
```

where a is the function name, b is the parameter list (as found in PASCAL, BBC BASIC, and so on), and c is a list structure containing the main body of the function.

We are now in a position to define our very first function. Suppose we frequently wanted to multiply numbers by the value 8. We could just write:

```
(TIMES 8 N)
```

each time, where N is the number to be multiplied. Instead let's define a function to do the same thing:

```
(DEFUN TIME8 (N) (TIMES 8 N) )
```

Here, the function TIME8 takes its argument N and uses the standard function TIMES to multiply this number by 8 and return a value. So the expression:

```
(TIME8 11)
```

would now give the result 88.

Before we can proceed further with functions, we must examine an important programming concept: the condition. Its importance can be clarified by imagining BASIC without IF . . . THEN statements!

The LISP condition statement is in the form of a function:

```
(COND (Condition1 Expression1)
      (Condition2 Expression2)
            :          :
      (ConditionX ExpressionX) )
```

First of all, note that this function has been spread over a number of lines to fit it all in. This is quite normal in LISP, which knows when you have finished an expression, because you will have closed the final bracket.

```
(DEFUN EQUAL (A B) (COND
   ((EQ A B) T)
   ((OR (ATOM A) (ATOM B)) NIL)
   ((EQUAL (CAR A) (CAR B)) (EQUAL (CDR A) (CDR B)))
   (T NIL)))
(DEFUN FIND_CAR (X L) (COND
   ((NULL L) '(NO NUMBER))
   ((EQUAL X (CDAR L)) (CAR L))
   (T (FIND_CAR X (CDR L)))))

(SETQ CARS '((JONES ABC123X) (SMITH XYZ789Y)
            (BLOGGS ACE99L) (BLYTHE JBB1))
(PRINTC (FIND_CAR '(ACE99L) CARS))
```

Search Procedure
This simple data retrieval program demonstrates the use of the CDR and EQ functions. A car registration number entered by the user will, if included in the program's database, return a list of two elements — the registration number and the surname of the car's owner

The COND function is very much like a PASCAL CASE statement (see page 1046). Condition1 is evaluated, and if it is true, then Expression1 will be used to return a result. Otherwise, Condition2 is tested, and so on. Some implementations of LISP (including Acornsoft LISP) require that at least one of the conditions (known as 'predicates') evaluates to true. This is easily handled by adding a final condition of the form:

```
(COND (Condition1 Expression1)
      (Condition2 Expression2)
            :          :
      (T          Final expression) )
```

Here, the character T is used to represent True; so the final expression (if reached) will always be evaluated. In LISP:

```
T = True = Not Zero
```

and:

```
F = False = Zero = ( )
```

The latter is the empty list. We can now define a slightly more complex function.

One very useful feature found in most dialects of BASIC is ABS, which returns the positive value of its argument. In other words, if its argument is positive, then it will be left unchanged; otherwise it will be negated. Our LISP function will have to be of the form:

```
(DEFUN ABS (X)
   (function body) )
```

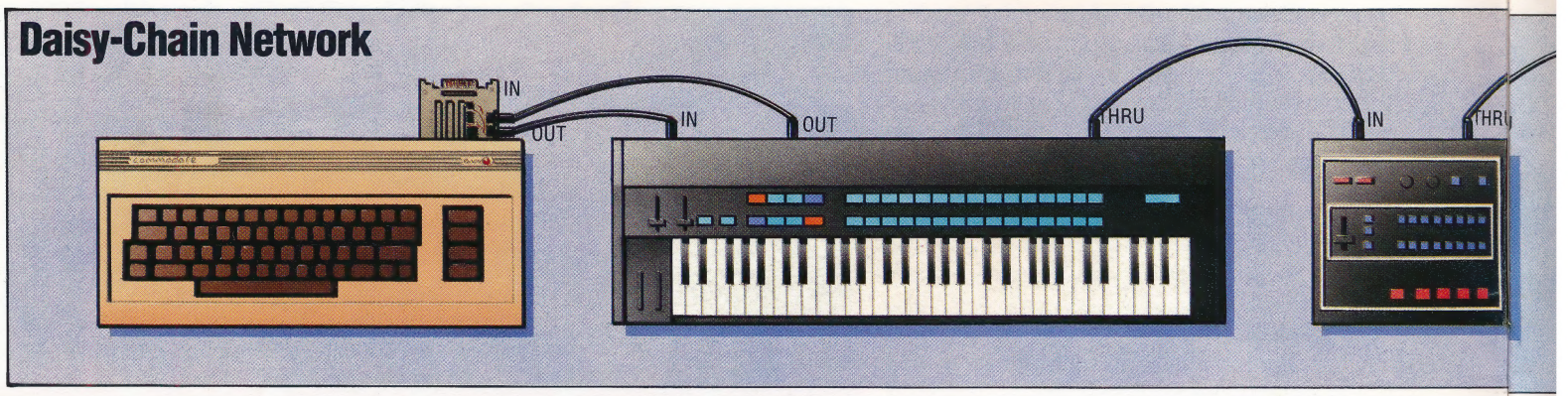
where X is the integer argument. Using our new condition function, we can then write the complete ABS function as:

```
(DEFUN ABS (X)
   (COND ( (MINUSP X) (MINUS X) )
         ( T          X         ) ) )
```

We've used two new functions here. MINUSP is a test function that returns the value if its argument is a negative number, and false otherwise. If the condition evaluates to T, the MINUS function negates its argument. If this isn't the case, the T condition (which is always true) will return the original value X.

Test Functions
We list below several other test functions that are frequently used in LISP

(OR arg1 arg2 . . .)	Returns True if at least one of its arguments is True
(ONEP arg)	Returns True if arg=1
(ZEROP arg)	Returns True if arg=0
(LISTP arg)	Returns True if arg is a list and False if it is an atom
(ATOM arg)	Returns True if arg is an atom and False if it is a list
(LESSP arg1 arg2)	Returns True if arg1 < arg2
(GREATERP arg1 arg2)	Returns True if arg1 > arg2
(EQ arg1 arg2)	Returns True if arg1=arg 2



Daisy, Daisy . . .
The most usual method of connecting MIDI-compatible instruments is to daisy-chain them together. Most instruments have MIDI THRU sockets (in addition to MIDI IN and MIDI OUT). These effectively allow the formation of a bus-type system, in which all the instruments are connected to the bus but only respond to messages on particular channels

RULES OF COMPOSITION

Having constructed the MIDI interface board, we now turn our attention to the equally important aspect of software, which will enable us to play 'real-time' music. We look at information vital to programming, examining the meaning of byte format values transmitted over the MIDI line.

We have already seen how a byte of data is transmitted via the MIDI to a receiving instrument (see page 1035). Now we will determine the format of the byte(s) needed to communicate the necessary information.

A single musical 'event' is transmitted as a group of bytes called a 'message'. Most messages are between one and three bytes long, with the exception of 'system exclusive' messages (discussed later), which may be any number of bytes long. Each message begins with a byte that has its most significant bit (MSB) equal to one, followed by the rest of the message bytes, all of which have their MSBs set to zero. A byte with its MSB set to one is called a *status* byte; the others are *data* bytes. MIDI messages are divided into two basic types – 'channels' and 'systems'.

The need for *channel messages* arises from the more usual daisy-chain method of interconnection used in small systems. In these, each unit receives all the data sent by the master transmitting unit.

Channel messages are used to transmit information that is not necessarily intended for all the units in a system. Most MIDI receivers may therefore be assigned a *channel number*, from one to 16.

Channel messages have status bytes with values in the range $80 to $EF inclusive, and have one or two data bytes. The channel number is encoded into the four least significant bits (the least significant hex digit) of the status byte with channel one represented by $0 and channel 16 by $F. The remaining three bits determine the type of message that follows.

A special feature of channel messages is that the status byte need not be sent if it is the same as the previous status byte. In effect, the current, or *running*, status remains in force until another status byte is received. The exception to this is when a system real-time message interrupts the current status temporarily.

This is particularly useful for communicating many consecutive note on/off messages, since a *note on* message with a velocity (how hard a key is pressed) of zero is equivalent to a *note off* message. Using the same status (note on) for a series of note on/off messages results in a saving of memory space and transmission time.

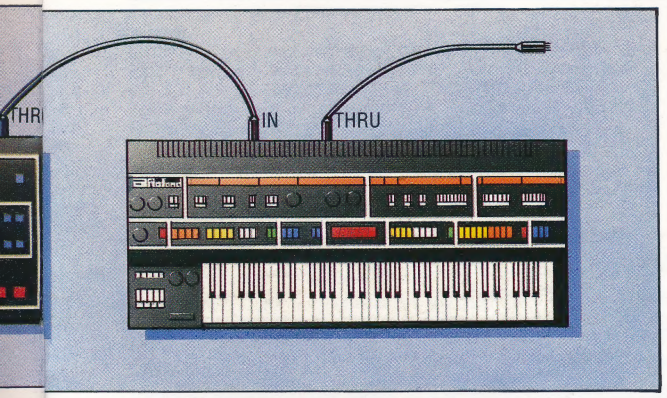
Voice assignment in synthesisers is the process of directing a note message (from either the MIDI or the instrument's keyboard) to one of the synthesiser voices currently available to actually produce the note. For example, a six-note polyphonic synthesiser is said to have six voices. To control the response of the instrument to channel messages, a number of modes are selectable by MIDI *mode messages*. A mode message should be ignored by a receiver that is incapable of operating in the requested mode.

MIDI MODE MESSAGES

Omni mode is the simplest mode, in which the receiver responds to all channel messages irrespective of the MIDI channel number encoded into the messages. When omni mode is turned off, the unit will respond only to messages sent on specific MIDI channels.

Most polyphonic synthesisers have a limited number of voices (usually six or eight), which means that certain action must be taken if a note is requested when all the voices are already assigned by previous note messages.

Usually the voices are assigned in strict rotation so that the least recent note is turned off to make the voice available for the new note. This mode of



operation is selected by the *poly mode on* message. A *mono mode on* message directs the receiver to assign each of its voices monophonically to one of a group of consecutive MIDI channels starting from the synthesiser's original (basic) channel. The mono mode message is sent on the basic channel and its second data byte specifies the total number of channels required.

If omni mode is on, a mono message simply directs the receiver to assign one voice monophonically to MIDI channel messages on all channels.

System messages are not encoded with channel numbers in their status bytes. Therefore, the status byte is received by all the units in a system. System messages fall into three categories: common, real-time and exclusive.

System common messages have status bytes from $F1 to $F7. They consist of the status byte followed by 0, 1 or 2 data bytes.

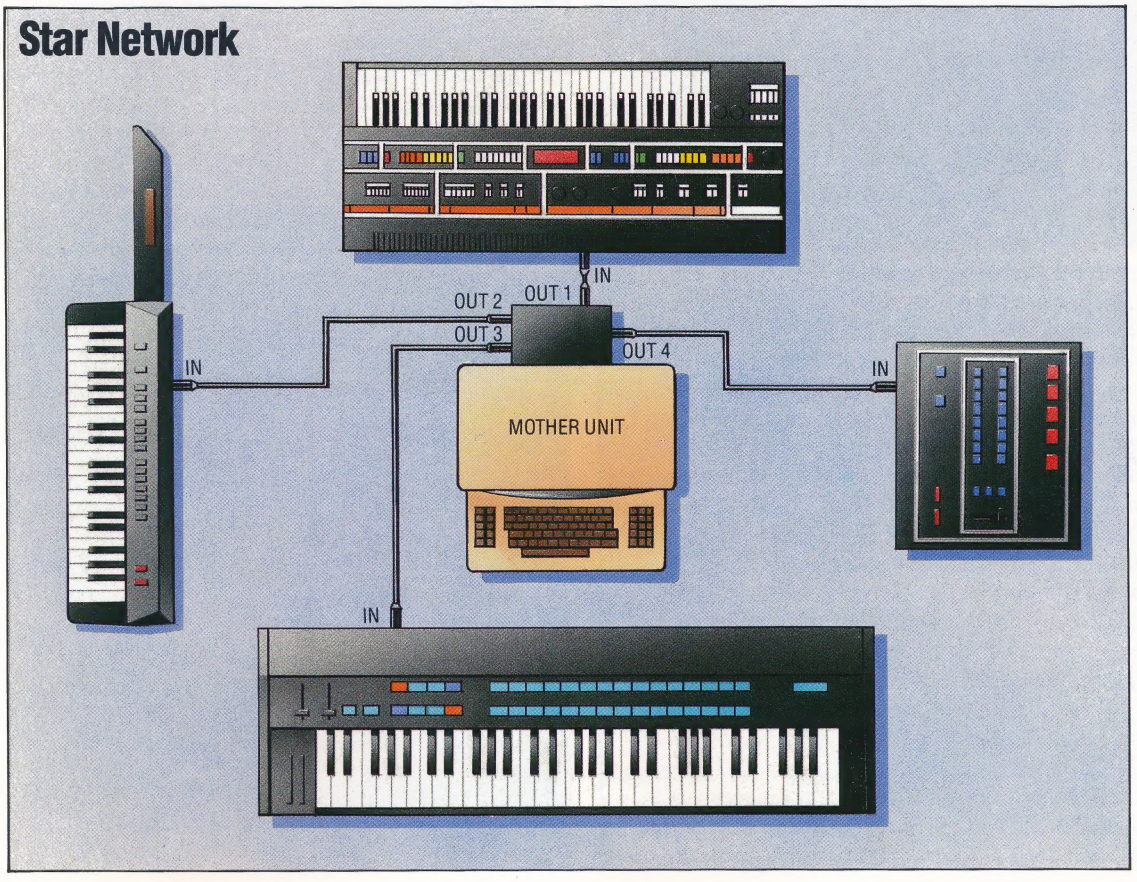
System real-time messages are intended for all units in the system. They have status bytes from $F8 to $FF and are used mainly by drum machines and sequencers to synchronise their own internal sequences to the clock in the master transmitter. Most synthesisers will ignore these messages unless they have some form of internal sequencer with the capability of MIDI synchronisation.

System real-time messages differ from other messages in that they all consist of one status byte only and no data bytes. As a consequence, they are allowed to be sent at any time, even if they interrupt messages of other types.

System exclusive messages begin with status byte $F0, followed by any number of data bytes, and are terminated by either the end-of-exclusive status ($F7) or any other status byte. The first data byte is a manufacturer's identification (ID) code. If this does not match the receiving unit's ID, the remainder of the message should be ignored.

The messages are intended for transferring data between instruments of a similar type, which would be meaningless to other units. The principal type of data transferred is 'patch program' data for synthesisers. (The transmission of a patch program in this way is not to be confused with channel message $Cx, which selects between patch programs already stored in the receiver's memory.)

However, the type of data transmitted by system exclusive messages is left to the manufacturers provided they have been assigned a valid ID code.



KEVIN JONES

Star Performance
Early implementations of MIDI did not allow channel selection, so all synthesisers would try to respond to messages on all channels — rendering the system almost useless when such devices were daisy-chained. These early MIDI units had to be connected in a star formation around a mother unit, which incorporated circuitry so that each channel could be obtained from a separate OUT socket

Channel Voice Messages

Status	Data1	Data2	Description	Notes
$8x	p	v	Note off p = pitch v = off velocity	1,2
$9x	p	v	a) v>0 Note on p = pitch v = velocity	1,2
			b) v = 0 Note off p = pitch	1,2
$Ax	p	pr	Polyphonic after-touch p = pitch pr = pressure	1,3a
$Bx	n	c	a) n <$7A Control change n = controller number c = controller value	4
			b) n=$7A Local control on (c=$7F) off (c=0)	5
			c) n=$7B, c=0 All notes off	6
			d) n=$7C, c=0 Omni mode off (all notes off)	6
			e) n=$7D, c=0 Omni mode on (all notes off)	6
			f) n=$7E Mono mode on (all notes off) c = number of channels	6,7
			g) n=$7F, c=0 Poly mode on (all notes off)	6
$Cx	pp	–	Patch Program Select pp = program number	
$Dx	pr	–	Channel pressure pr = pressure value	3b
$Ex	l	m	Pitch Wheel Change l = 7 LSBs m = 7 MSBs	8

Notes:

Except where otherwise specified, all data bytes may have any value between 0 and $7F. The channel number is represented by x, the least significant (hex) digit of the status byte (x=0 specifies channel 1 and x=$F specifies channel 16).

1. p is the pitch of the note in semitones. Middle C is pitch number $3C and so all Cs are multiples of $0C (decimal 12). The standard 88-note piano keyboard ranges from $15 to $6C.

2. Velocity values range from $01 to $7F. Keyboards without velocity sensors should use a default velocity of $40. A 'note on' message with a velocity of 0 is equivalent to a 'note off' message with a default velocity of $40. This will generally be the method used by simpler keyboards having no release velocity sensors — thus taking advantage of the running status feature of channel messages.

3. 'After touch' is the amount of pressure exerted on a key after it has been struck. It is generally used to introduce modulation effects without having to operate a modulation wheel. There are two types of pressure sensor, resulting in two different types of message.
a) Individual or polyphonic after touch requires individual pressure sensors for each key and affects only the note corresponding to the key pressed. Consequently, a pitch number must be sent in addition to the pressure value. (Its implementation also requires that separate modulation circuits are provided for each voice).
b) Channel pressure is produced by only one pressure sensor, which is affected by all keys equally. The pressure that is transmitted corresponds to the maximum pressure (all keys held down at one time). The resulting effect is applied to all voices simultaneously.

4. These messages are sent by controllers separate from the keyboard. Controller numbers from $0 to $1F are continuous controllers having control values (c) from $0 to $7F. These are equivalent to computer potentiometer joysticks and may in fact be joysticks or other devices such as wheels, foot pedals and breath controllers. Controller numbers from $20 to $3F are optionally used to send an extra seven least significant bits for controllers $0 to $1F, if very high resolution is required. Controller numbers from $20 to $3F are optionally used to send an extra seven least significant bits for controllers $0 to $1F, if very high resolution is required. Controller numbers from $40 to $5F are switch controllers (such as sustain pedals, portamento, on/off switches, and so on) with either c set to zero (off) or $7F (on). These are equivalent to the more usual type of switch joystick used by computers. Controller numbers from $60 to $79 are undefined, and those from $7A to $7F are reserved for channel mode messages. The controller numbers are not required to be assigned to specific physical controllers, except that the modulation wheel is usually assigned to controller number $1.

5. Local control is optionally used to break the internal link between a unit's input devices (usually a keyboard and associated controllers) and its sound-generating circuitry, so that the keyboard (for example) only sends data to MIDI OUT and the sound-generating circuits respond only to data received at MIDI IN.

6. The implementation of 'all notes off' is optional, and these messages should not be used instead of individual 'note off' commands to turn off a number of notes. The reasoning behind this is unclear and renders the commands virtually useless, where they might have been used to turn off notes still sounding if a sequence were stopped before its endpoint.

7. The third byte of the mono mode message specifies the total number of channels required. If this number is zero, the number of channels equals the number of voices available in the receiver.

8. The pitch wheel differs from other controllers in that it can have positive and negative values. The centre position is sent as l=$0 and m=$40. Many receivers have only seven bits of resolution and will respond only to m, ignoring the value of l

System Messages

Status	Data1	Data2	Description	Notes
$F0	any number		System exclusive	1
$F1			Undefined	
$F2	l	m	Song position pointer l=7 LSBs m=7 MSBs	2
$F3	s		Song select s=song number	
$F4			Undefined	
$F5			Undefined	
$F6			Tune request	3
$F7			End of system exclusive	1
$F8	—	—	Timing Clock	4
$F9			Undefined	
$FA	—	—	Start	4
$FB	—	—	Continue	4
$FC	—	—	Stop	4
$FD			Undefined	
$FE	—	—	Active Sensing	5
$FF	—	—	System Reset	6

Notes:

The messages $F8 to $FF are the system real-time messages and may be sent at any time (even during transmission of other messages).

1. System-exclusive messages may have any number of data bytes terminated by 'end of system-exclusive' ($F7) or any other status byte.

2. This message is used to arbitrarily preset the song position pointer, which is an internal register containing the number of beats (1 beat = 6 MIDI clocks) since the start of the sequence (song).

3. This is used to request analogue synthesisers to tune their oscillators.

4. These messages are used to synchronise master and slave sequencing units. The system clock is set at a rate of 1/24th of a quarter note. The 'continue' command differs form 'start' in that it restarts a sequence from the pointer at which it was stopped, rather than from the beginning.

5. Active sensing is optionally used as a dummy message when there is no other activity on MIDI. If used, it should be sent so that no longer than 300ms pass without activity. A receiver should act normally if it never receives active sensing; otherwise, if no activity is present for longer than 300ms, it should turn off its voices and return to normal operation.

6. This message is used to initialise the entire system to the power-up condition. It should not be sent automatically on power-up, to avoid the possibility of two units endlessly resetting each other.

Any messages not listed above should not be sent, and should be ignored by receivers. In particular, care should be taken to suppress spurious transmissions on power-up

A LITTLE PEACH

CHRIS STEVENS

Although the Apricot F1e is intended primarily as a business machine, it is likely to be very appealing to the educational and small business markets as well. With its entry price of £684, the F1e comes well within the price range of the serious home computer user too.

The basic setup of the F1e will be easily recognisable to those familiar with Apricot computers. Sold as a package, it includes the F1 keyboard, the computer unit itself with a single-sided single-density disk drive, and an eight-inch 'green screen' monitor. As with the Apricot Portable (see page 1109), the company has included its infra-red technology, which removes the need for a large number of trailing leads. Both the keyboard and the optional mouse control the computer via infra-red beams that are detected by sensors on the front of the F1e. Reflecting the trend towards smaller computers (initiated by the Apple Macintosh), which do not take up large amounts of desk space, the F1e is remarkably narrow, measuring only 200mm (8in) across, although this is somewhat offset by the length, which is around 425mm (17in).

The keyboard is in the standard Apricot layout – the keys are flush and similar in appearance to those on the Sinclair QL. The keyboard is not particularly good for touch-typing, and therefore not very well suited to word processing applications. The keys themselves, while adequate for a modern business machine, rattle a bit and might raise suspicions concerning their long-term reliability. Like the Apricot Portable, there are four buttons at the top of the keyboard: 'Reset', for cold starts, 'Repeat Rate', allowing you to vary the rate at which characters are repeated when a key is held down, 'Set Time' and 'Keyboard Lock'.

The computer unit has a single Sony 3½in disk drive, built into the front, which is becoming increasingly popular with manufacturers and has therefore justified ACT's early use of this format for its machines in more ways than one. While some other companies are having difficulties transferring their wide software bases onto the 3½in disks, all Apricot software is conveniently compatible. Users wishing to purchase the F1e therefore have no need to worry about a lack of suitable software. Furthermore, the single-sided single-density disks used with the F1e can hold a maximum 315 Kbytes of information, more than their 5¼in counterparts.

Other features on the front of the computer unit include a series of LEDs indicating power, Caps Lock, scroll and drive on/off. Beneath these are the infra-red light beam detectors.

On the right-hand side of the computer is a 60-way expansion bus, through which a wide range of expansion boards, or an extra disk drive, can be added. This slot makes the F1e expandable to the capacity of the Apricot XI by adding the MSD expansion system, which provides 10 megabytes on hard disk.

The rear panel, while not having many interfaces, is sufficiently well-designed to allow the more common peripherals to be fitted. On the far-left is a standard 25-way D connector that accommodates an RS232 serial interface. Next to this port are two monitor sockets. The first is a nine-pin adaptor that provides an RGB signal for colour monitors (although the monochrome ACT monitors that are available also plug into this socket). To the right of this is a composite video socket for other types of monitor.

The £684 price of the F1e does not include a monitor. The computer does not have an RF adaptor enabling it to send a signal to a standard television set; however, there is an adaptor available to provide such a signal. Also, since a

Good-Looker
The Apricot F1e has inherited the good looks of its relations. The integral 3½ in disk drive, standard MS-DOS operating system and bundled software packages, together with a price tag of under £700, make the F1e a serious challenger to the BBC Model B and B+ systems, which are currently favoured in educational circles and by the serious home computer user

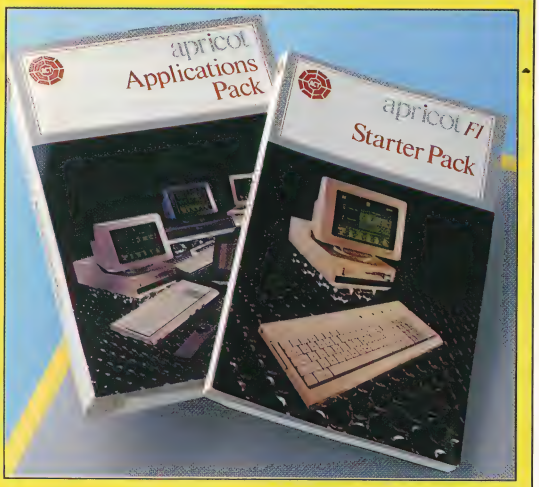
monitor is not fitted as standard to the F1e and many monitors have their own power input, the monitor sockets do not automatically provide power lines, and the computer itself is not capable of providing such a source. Therefore, an external 17v power supply has to be fitted to the computer to enable it to run one of ACT's monitors.

Although this seems logical enough, the fact that ACT has made the decision to remove the usual trailing wires by providing infra-red communications is at odds with its decision to provide an external power supply to run its monitors. This is especially so since other models in the range (like the company flagship ACT Apricot) do not require such a device. The power supply can be tucked out of the way, especially if it is set up in a permanent position, but you are still left with the impression that this aspect of the machine was hastily thought out.

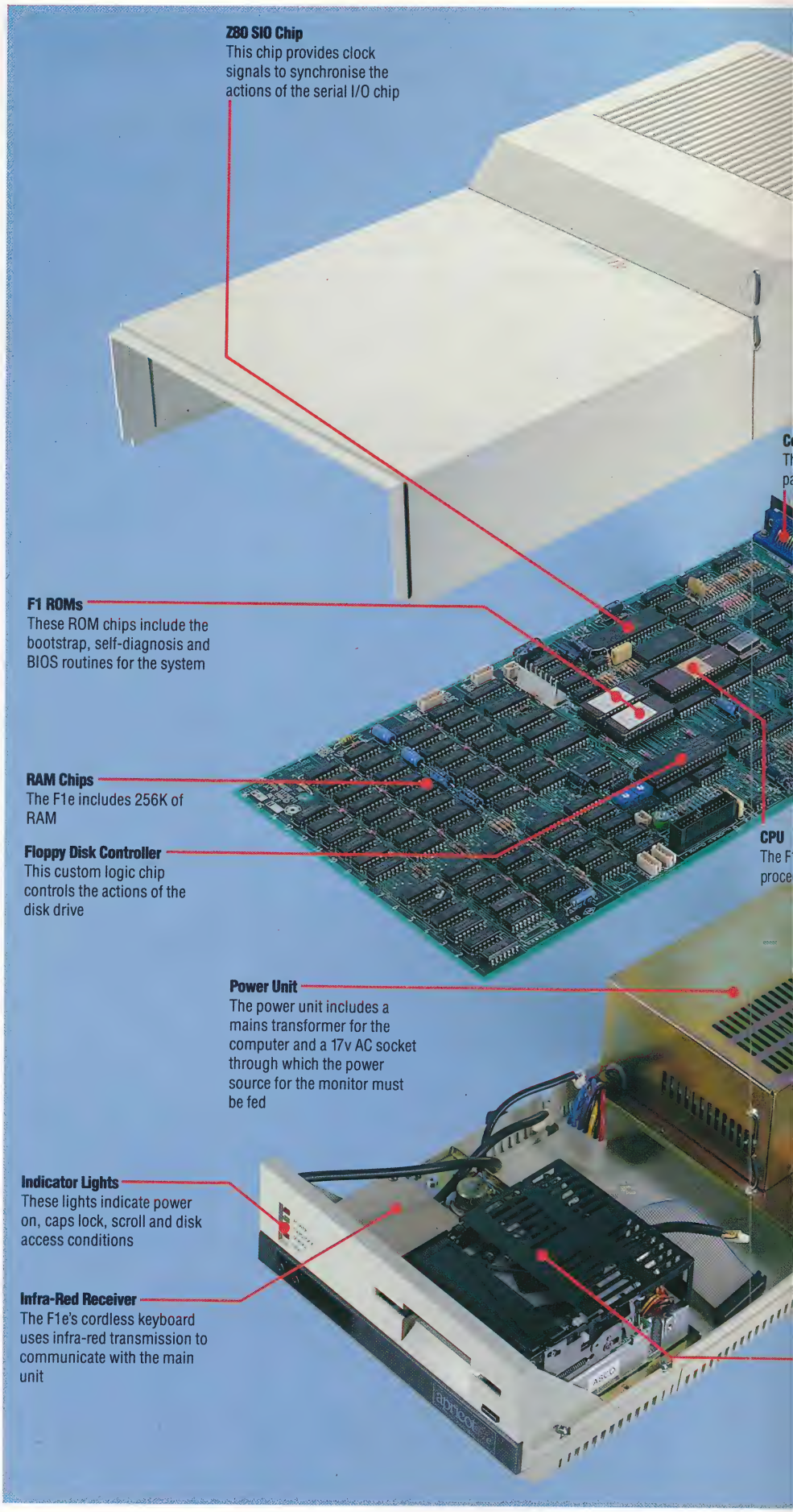
Inside the computer, there is a further expansion slot for extra boards. The F1e's board, in common with others in the Apricot range, is elegantly designed and is entirely protected from the disk drive and power transformer (which can generate harmful temperatures) by a metal casing that acts as a heat sink.

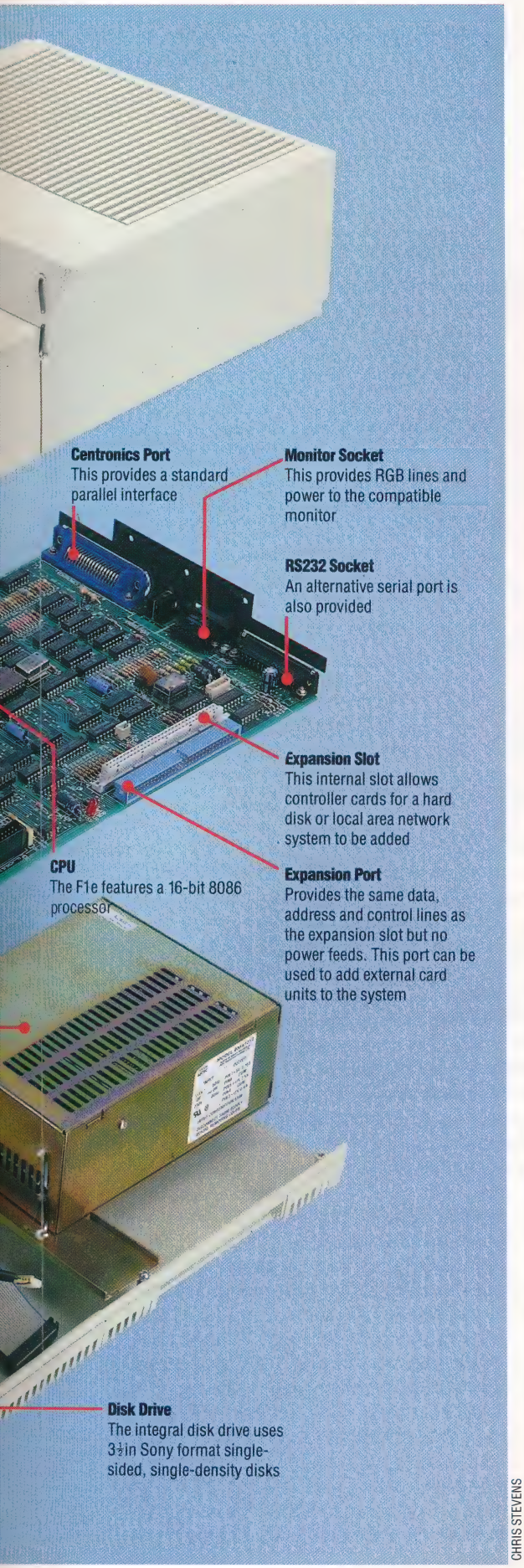
The F1e is based around the 16-bit 8086 processor and runs under the popular MS-DOS operating system. Apricot computers, however, have always had their own icon-based management system, which means, in effect, that you will have little to do with the operating system directly. The management system on the F1e is known as Activity, and it is with this that users will become most familiar.

Activity is an object-oriented program. This means that rather than issuing a series of commands to the computer, such as one to LOAD a file into memory, you merely indicate (generally by selecting an icon from a series presented on the



Super Software
The free software provided with the F1e includes a word processor, electronic notebook and spreadsheet. Although the first two programs (SuperPlanner and SuperWriter) are relatively unsophisticated, the spreadsheet (SuperCalc) has many features, such as windowing, that one would expect to find in an expensive spreadsheet application


Z80 SIO Chip
This chip provides clock signals to synchronise the actions of the serial I/O chip
F1 ROMs
These ROM chips include the bootstrap, self-diagnosis and BIOS routines for the system
RAM Chips
The F1e includes 256K of RAM
Floppy Disk Controller
This custom logic chip controls the actions of the disk drive
CPU
Power Unit
The power unit includes a mains transformer for the computer and a 17v AC socket through which the power source for the monitor must be fed
Indicator Lights
These lights indicate power on, caps lock, scroll and disk access conditions
Infra-Red Receiver
The F1e's cordless keyboard uses infra-red transmission to communicate with the main unit


Centronics Port
This provides a standard parallel interface
Monitor Socket
This provides RGB lines and power to the compatible monitor
RS232 Socket
An alternative serial port is also provided
Expansion Slot
This internal slot allows controller cards for a hard disk or local area network system to be added
CPU
The F1e features a 16-bit 8086 processor
Expansion Port
Provides the same data, address and control lines as the expansion slot but no power feeds. This port can be used to add external card units to the system
Disk Drive
The integral disk drive uses 3½in Sony format single-sided, single-density disks

CHRIS STEVENS

screen) which action you wish to perform; the computer will do the rest.

Most of these systems depend on the use of a mouse to move the cursor around on the screen, but on the F1e system, the numeric keypad can be substituted. The figures 1 to 9 are arranged in three rows of three keys. Taking the keypad as a compass (excluding the centre key) there are eight directions in which the cursor can be moved. Thus pressing the middle key of the top row (8) will move the cursor straight upwards to an icon in that direction, whereas pressing 3 (the right position on the bottom row) will move the cursor diagonally down to the right. To select an icon you have to press the Enter key on the calculator keypad.

Among the suite of programs included on the Activity system disk is a tutorial program to help accustom you to the system. It also explains the usages of the icon and fount editors (with facilities to load your own character sets whenever you choose) and the system configurator, which allows you to set up the computer for whatever specialised peripherals you may be using.

THE BUNDLED SOFTWARE

As you might expect from the F1e, which is intended primarily as a business machine, it has several software packages included in an applications pack. SuperWriter is a word processing program based around the widely used WordStar, but without the latter's breadth of formatting capabilities. SuperPlanner is described as an 'electronic notebook' that allows you to 'forward-plan' your activities. Included within this program is an address book, a calendar, a diary and a small filing system. Although superficially resembling a database, SuperPlanner does not contain the sophisticated search and retrieval techniques present on a true database.

The final package bundled with the F1e is SuperCalc. This is a spreadsheet for financial and accounting applications. This is perhaps the most complete of the three packages, containing a wide number of commands allowing windows, text justification and a number of other features you would expect from a professional spreadsheet package.

In reducing the price of the F1e, ACT has let the computer industry know that it intends to make a concerted push into the educational market. This is emphasised by the fact that along with the announced price cut, ACT released a new product called B-TRAN, which enables the computer to run virtually all programs written in BBC BASIC.

The reduction in the price of the F1e will certainly improve the sales of the computer, particularly in the educational and small business markets, which have so far been dominated by the BBC Micro. Although it is not likely to dominate the home computer market in the same way the Sinclair Spectrum has, the F1e is nevertheless an excellent purchase for those interested in buying a low-priced machine that can be expanded to the power of a fully-equipped business computer.

APRICOT F1e

PRICE
£684 inc VAT (monitor extra)

DIMENSIONS
425×200×105mm

CPU
Intel 8086 processor running at 4.7 MHz

MEMORY
256 Kbytes standard

SCREEN
Text resolution 132×50 or 80×25 characters, or a maximum graphics resolution of 800×400 pixels

INTERFACES
RS232 port, Centronics interface and RGB and composite video sockets

DISK DRIVE
Single 315 Kbyte 3½in disk drive

OPERATING SYSTEM
MS-DOS, CP/M-86 and Concurrent CP/M

DOCUMENTATION
The applications pack manual is thorough and consistent with ACT's usual high standard, although the starter manual is a little thin on detailed information about the machine itself

STRENGTHS
At the price, the F1e seems to be a bargain and could have a great impact on the small business and education market

WEAKNESSES
The flat keyboard is not up to the standard of the ACT Apricot. Although containing a fast processor, the F1e is not as quick as some comparable machines

PROMPT

This is a character or message displayed on a television or monitor screen, when a computer is expecting some input from the user. Often the *prompt* will indicate what kind of data it is expecting to receive (for example, whether the computer expects numeric or alphabetic input).

Prompts can also tell us something about the status of the computer. When we are running a computer under one of the popular disk operating systems, such as CP/M or MS-DOS, the prompt will appear as A>. This tells us that the currently logged drive is A and that any command that is sent for execution (unless specified otherwise) will be expected to operate on the disk in that drive.

PROTOCOL

When data is transmitted between two devices, there must be a set of rules that govern the exchange of information. This set of rules is known as the *protocol.* The kinds of parameters that are set by the protocol include the transmission speed, the amount of data to be sent at once and information about how to identify the receiving device, how the data is received and in what form it is to be transmitted. The protocol will also dictate how the data is to be checked for errors, as well as define the method of requesting retransmission in the event of an error.

A system of protocols is used when accessing public databases. In this case, the protocol sets the baud rate, the method of transmission (ASCII codes) the start and stop bits — which indicate the boundaries of a separate piece of information — and the parity checking, which informs the target computer whether the information has been received correctly.

PULSE

A *pulse* is defined as a rise (usually relatively large and rapid) of the amplitude of an electrical current, sound or electromagnetic wave, followed after a short period of time by a corresponding drop in the amplitude. In electronics, the change in amplitude usually refers to a change in the voltage.

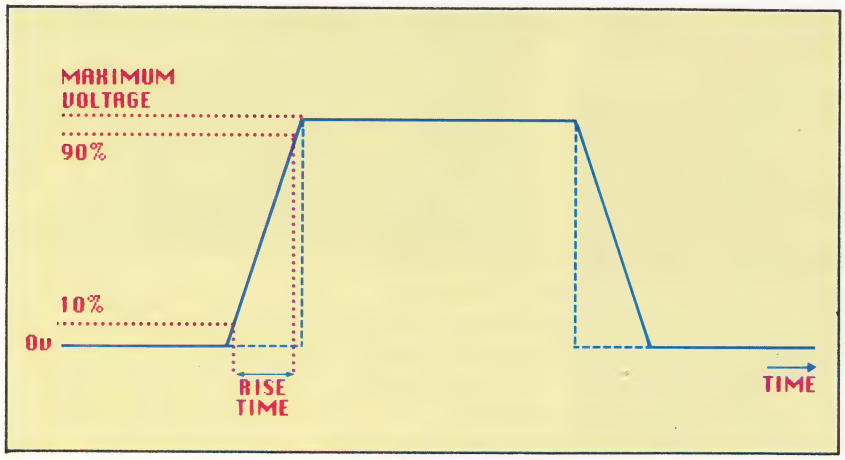
The parameters governing the shape of a pulse are as follows. The *rise time* is the period taken for the amplitude to rise (this would be almost instantaneous in the case of the pure square wave) until it reaches the *pulse height.* At this point, the amplitude will be maintained until it begins to fall back to the original value. This section is therefore known as the *decay time.* The time between the beginning of the rise of the amplitude and its subsequent return to the original value is known as the *pulse width.*

The generation of pulse waves is fundamental to the working of most computers. An example of the use of sharply defined pulses within a computer is the *system timer.* Here, a highly accurate 'clock' is required in order to synchronise the system operations. The rise and delay times, which trigger the actions of the processor, are kept extremely short in order to ensure that the cycles are kept as regular as possible.

The invention of the transistor had a fundamental effect on the development of digital computers based around the pulse wave. A property of many transistors is that they will not allow a current to pass unless the voltage (potential difference) reaches a certain value. The transistor will then transmit the current until the voltage falls once more below that value, when transmission will cease instantly. The resultant effect is a *pulse wave.*

Another major use of the pulse in microelectronics is in the production of synthesised sound. The square wave is an important component of many electronic instruments, and is used to add a 'harder' tone to many sounds.

Rise And Shine
This diagram shows the development of a pulse wave. Note that the change from 0v to the maximum does not occur instantaneously, but rises over a period of time. The rise time is measured by how long it takes for the pulse to change from 10% to 90% of its maximum (and vice versa for the decay time)

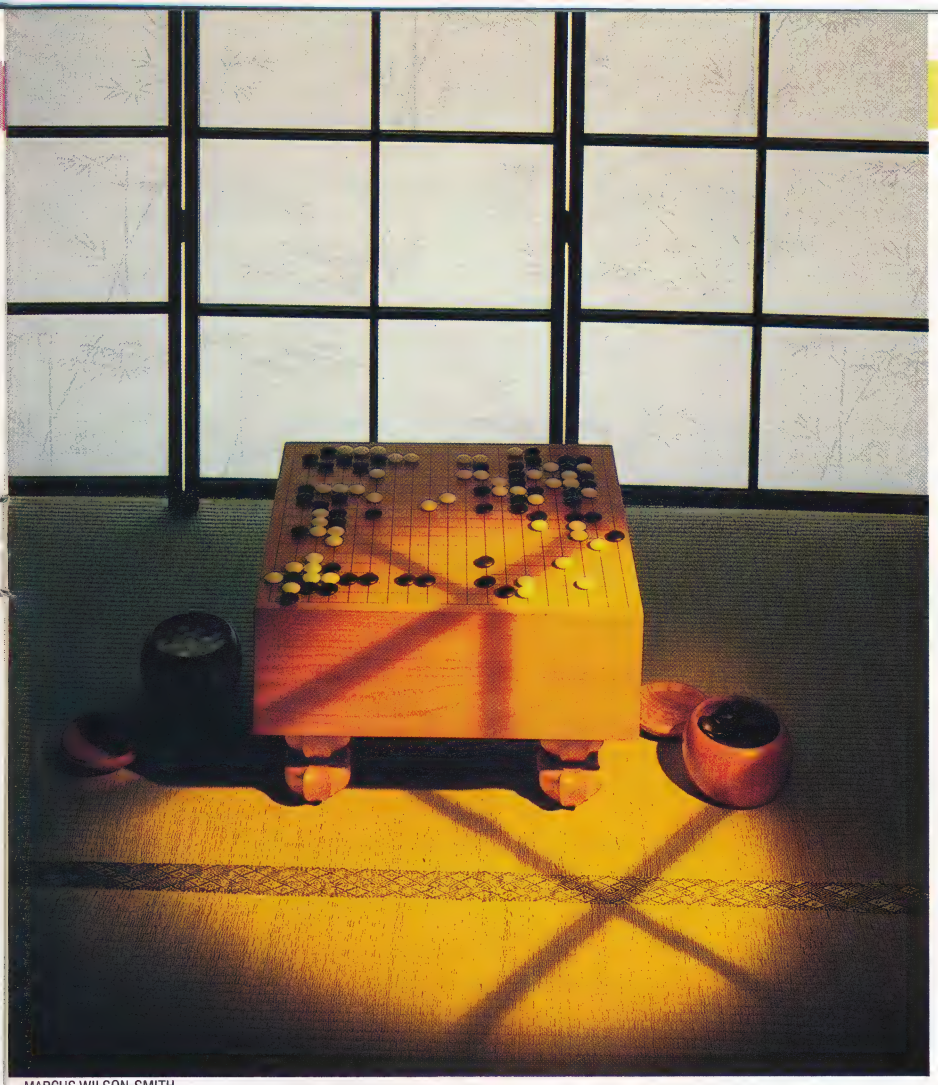


PUNCHED CARD

Although they are now largely obsolete, *punched cards* were once widely used for inputting data into computers. A standard card measures 18.75 by 8.25cm and is divided into 12 rows and 80 columns, though there have been many variations. Combinations of holes represent particular characters. These are sensed by a 'card reader', which translates them into digital signals that the computer can understand.

The keyboard, magnetic tape and disks have, for the most part, replaced the punch card, but it has left some influence in the computing field. The programming language COBOL, for instance, will not accept a line over 80 characters in length, a reminder of the 80-column cards for which the language was developed.

The notion of storing data on punched cards is one of the oldest ideas in computing. Herman Hollerith, inventor of the familiar punched card and founder of the company that became IBM, developed his system at the end of the last century. Before him, Charles Babbage, often referred to as the father of computing, believed that the method was ideal for storing data in the 'analytical engine' he invented, as long ago as 1822. The earliest example, however, stems from 1802, when Joseph Jacquard built a loom that recorded the patterns of fabrics on what were, for all intents and purposes, punched cards.



MARCUS WILSON-SMITH

Opening Move
Traditionally the game of Go is played on a wooden board etched with a 19 by 19 grid of intersecting lines. In our computerised version of the game, we have reduced the board size to a 15 by 15 grid to allow the board to be easily displayed on a VDU screen

SET-UP SHOTS

When writing a program for a game like Go it is always best to begin with the I/O routines. We start our project by outlining these routines for the BBC Micro — initialising variables, producing the Go board on screen and providing a list of the variables and their usages.

The Go program will be presented in four separate versions: for the BBC Model B Micro, the Commodore 64, the Sinclair Spectrum and the Amstrad CPC 464/664 micros. Where possible, the line numbers in these versions have been kept consistent, but for various reasons extra lines have been needed to implement some routines on the different machines. This is especially evident where recursion has been used in BBC BASIC (recursion is not implemented on the other three machines).

Consequently, it is necessary to implement a user stack in order to keep the listings as similar as possible. When the text refers to routines and variables in the listing, these will apply to the BBC Micro version, though cross-referencing between the four listings is quite straightforward.

The following section of program deals with the initialisation of variables, creating the board display and providing the input routines. Lines 10 to 140 constitute the main body of the program. Before entering the game loop itself (lines 60 to 90), the routines PROCinitialise and PROCintroduction will be called.

PROCinitialise is only used when first running the program to DIMension the board (board%), initialise the cursor and so on. The board is DIMensioned to be a series of 255 bytes, creating a 15 by 15 playing area (rather than the usual 19 by 19 Go board).

There are a number of reasons for this, the most important of which is speed. A BASIC game-playing program cannot be expected to operate particularly quickly, but by reducing the size of the board, execution time will be cut by about one third without affecting the way in which the game is played. Of course, this also means that the board fits neatly into one page (256 bytes — one quarter of a Kbyte) of memory, allowing for a boundary made up of single squares. By ensuring that all board references are in the range 0 to 255, this border will encircle the entire board.

The variables initialised in lines 190 and 200 are constants, but by using the variable name instead of the actual numbers, modification is made much easier. For instance, if you wanted a two-player game, or the computer to play itself, you would only need to change the values of black% and white% in order to use all the routines normally. Based on these values, each board byte will be used in the manner shown in the diagram.

PROCintroduction is used at the start of each game, to set up the move counter (move%), the end condition (end%), and so on. This makes use of the routines PROCgame_init and PROCtitle_screen. The title screen is fairly elementary due to space restrictions for the listings, but you should feel free to modify this as much as you want. For instance, you could add the Japanese Go symbol, or possibly a few strains of Oriental music. If you'd like to add pieces of your own code, all line numbers from 5000 onward are free for this.

PROCprint_board, naturally enough, prints the board. FNint_to_ char is used from this routine to convert an integer board co-ordinate into the character co-ordinates used by the screen display; it is DEFined at line 2260.

FNinput and PROCmessage are two general-purpose routines for user input and message printing, based on the mess$ array initialised in PROCread_messages. This array holds 10 prompts or messages that the computer needs to display

during the game. The appropriate message is selected by passing the corresponding array element number, M%, to these two routines. Incorporating these routines, rather than just prompting and receiving input and output when needed, increases the flexibility of the program.

One problem you may have when typing in the Spectrum version is between lines 1480 and 1505. The underlined numbers in these lines refer to the Spectrum's symbol graphics. You'll get these by first pressing Caps Shift and 9 together, thereby entering graphics mode (the G cursor), and then typing the numbers shown. Press Caps Shift and 9 again to leave graphics mode at the end of the print statement. If the number is preceded by sh, then type the number while pressing a Shift key.

If you run the code as it stands, you'll get a title screen asking for a number of handicap stones, followed by the main game screen. The program will then loop indefinitely. The handicap number is ignored at present, but we'll add this and a few more general routines in the next instalment.

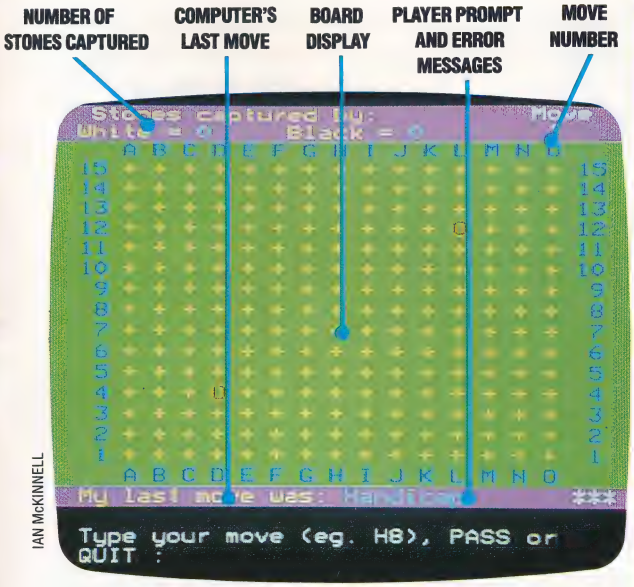
Go Game Variables

Variable	Usage
black%	Value 1: black piece in board byte.
board%	The start value for the 256–byte board in memory.
capture%(2)	Holds the number of stones captured by black and white.
colour%	Value 3. This is used for masking the colour bits in a board byte.
dir%(4)	This array holds the offsets necessary to move one square North, East, South and West.
liberty%	Value 8. Is used in group–searching routines to mark previously counted liberties.
marker%	Value 4. Is used in group–searching routines to mark previously counted stones.
move%	Holds the number of moves played in each game for display.
white%	Value 2: white piece in board byte.
atari1$	Holds either five spaces or 'Atari' if the computer's last move caused this situation.
atari2$	Holds either five spaces or 'Atari' if the player's last move caused this situation. Note: Atari is the act of placing a stone so as to leave one or more of your opponent's groups with only one liberty.
axis$	Part of the board display.
mess$	Holds all the I/O general messages, used by either PROCinput or PROCmessage.

Module One

BBC Micro:

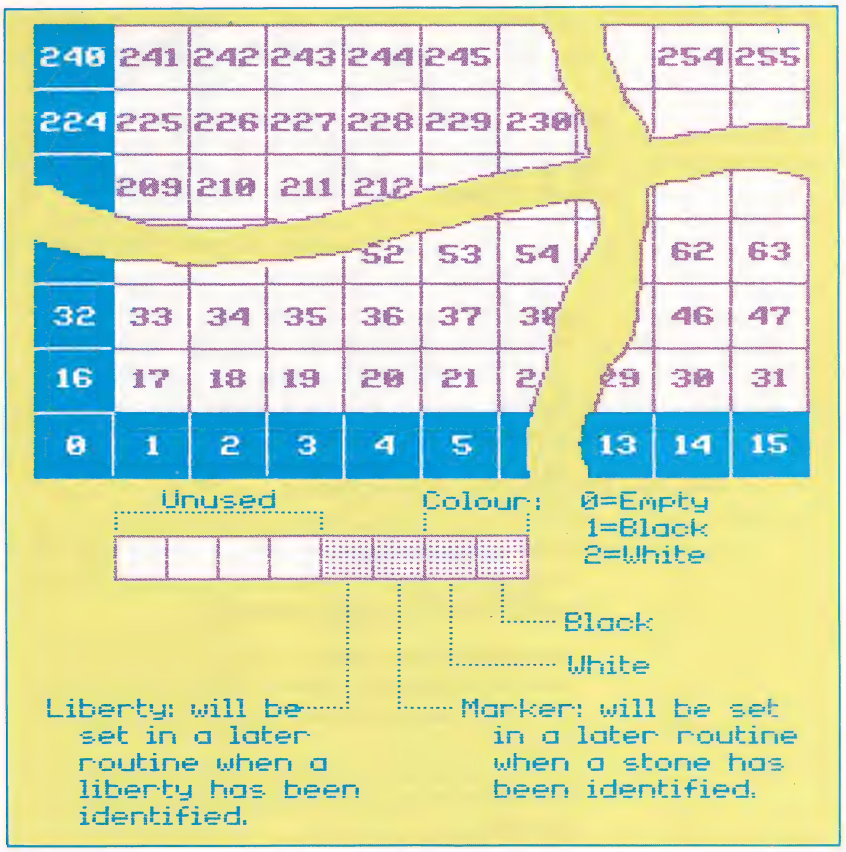
```
   10 MODE 7
   20 *FX14,6
   30 PROCinitialise
   40 PROCintroduction
   50 CLS:PROCprint_board
   60 move%=move%+1:REM white moves here
   70 IF end% GOTO 100
   80 move%=move%+1:REM black moves here
   90 IF NOT end% GOTO 60
  100 ans$=FNinput(21,9,1)
  110 IF ans$="Y" GOTO 40 ELSE IF ans$<>
"N" GOTO 100
  120 PROCmessage(22,5,"")
  130 PRINT:END
  140 :
  150 REM*****************************
  160 :
  170 DEF PROCinitialise
  180 LOCAL L%
  190 black%=1 : white%=2 : colour%=3
  200 marker%=4 : liberty%=8
  210 @%=2
  220 DIM board% 255
  230 VDU 23;8202;0;0;0;
  240 DIM capture%(2)
  250 axis$=CHR$130+CHR$157+CHR$132+"
A B C D E F G H I J K L M N O"
  260 PROCread_messages
  290 DIM dir%(4)
  300 RESTORE 340
  310 FOR L%=1 TO 4
  320   READ dir%(L%)
  330   NEXT
  340 DATA 16,1,-16,-1
  350 ENDPROC
  360 :
  370 REM*****************************
  380 :
  390 DEF PROCread_messages
  400 LOCAL M%
  410 RESTORE 460
  420 DIM mess$(9)
  430 FOR M% = 0 TO 9
  440   READ mess$(M%)
  450   NEXT
  460 DATA "O.K. THINKING..."
  470 DATA "Illegal entry: "
  480 DATA "Stone already on point: "
  490 DATA "Illegal. Ko on point: "
  500 DATA "Illegal. Suicide on point: "
  510 DATA "    O.K. GAME OVER."
  520 DATA ""
  530 DATA "  How many handicap stones m
ay I          have (2-9) ?"
  540 DATA "Type your move (eg. H8), PAS
S or        QUIT :"
  550 DATA "Do you want to play again (Y
/N)?"
  560 ENDPROC
  570 :
  580 REM*****************************
 1260 :
 1270 DEF PROCintroduction
 1280 PROCgame_init
 1290 PROCtitle_screen
 1300 ENDPROC
 1310 :
 1320 REM*****************************
 1330 :
 1340 DEF PROCgame_init
 1360 atari1$="     ":atari2$="     "
 1370 location%=0:move%=1
 1380 end%=FALSE
 1390 capture%(1)=0:capture%(2)=0
 1410 ENDPROC
 1420 :
 1430 REM*****************************
 1440 :
 1450 DEF PROCtitle_screen
 1460 CLS
 1470 PRINT TAB(12,4);CHR$145;CHR$154;"h
7+$  h7k4"
 1480 PRINT TAB(12,5);CHR$145;CHR$154;"j
5k5  j5j5"
 1490 PRINT TAB(12,6);CHR$145;CHR$154;"p
ssp  pssp"
 1500 PRINT TAB(9,10);CHR$134;"by  Marcu
s  Jeffery"
```


NUMBER OF STONES CAPTURED
COMPUTER'S LAST MOVE
BOARD DISPLAY
PLAYER PROMPT AND ERROR MESSAGES
MOVE NUMBER
Type your move (eg. H8), PASS or QUIT :

IAN McKINNELL

Running Commentary

The screen display above shows the layout of the various features of the game. In addition to showing the board, the program informs the player of the number of stones captured by both the computer and the player, the number of moves played so far in the game, and states what were the last moves made by the computer and its human adversary. The bottom portion of the screen is reserved for prompt and error messages to the player. As the game is developed we will see how the program detects and disallows illegal moves such as 'suicide' or any attempts to place stones on positions that are already filled


240 241 242 243 244 245
254 255
224 225 226 227 228 229 230
209 210 211 212
52 53 54
62 63
32 33 34 35 36 37
46 47
16 17 18 19 20 21
29 30 31
0 1 2 3 4 5
13 14 15
Unused
Colour:
0=Empty
1=Black
2=White
Black
White
Liberty: will be set in a later routine when a liberty has been identified.
Marker: will be set in a later routine when a stone has been identified.

```
 1510 PRINT TAB(1,13);CHR$133;"You will
play the";
 1520 PRINT CHR$135;"white";CHR$133;"sto
nes,  and"
 1530 PRINT CHR$133;" the  computer  (be
ing weaker!)  will"
 1540 PRINT CHR$133;" have the";CHR$129;
"red";CHR$133;"stones with a  handicap"
 1550 PRINT TAB(13);CHR$133;"advantage."
 1560 hand%=VAL(FNinput(20,7,1))
 1570 IF hand%<2 OR hand%>9 THEN 1560
 1590 ENDPROC
 1600 :
 1610 REM*******************************
 1720 :
 1730 DEF PROCprint_board
 1740 LOCAL P%,X%,Y%
 1750 PRINT TAB(0,0);CHR$133;CHR$157;CHR
$131;" Stones captured by:";
 1760 PRINT TAB(32,0);CHR$135;"Move"
 1770 PRINT TAB(0,1);CHR$133;CHR$157;CHR
$131;"White =";CHR$134;capture%(2);
 1780 PRINT TAB(16,1);CHR$131;"Black =";
CHR$134;capture%(1);
 1790 PRINT TAB(33,1);CHR$135;move%
 1800 PRINT TAB(0,2);axis$
 1810 FOR Y%=15 TO 1 STEP -1
 1820   PRINT TAB(0,18-Y%);CHR$130;CHR$1
57;CHR$132;RIGHT$(" "+STR$(Y%),2);
 1830   FOR X%=1 TO 15
 1840     P%=board%?(16*Y%+X%)
 1850     IF P%=1 THEN PRINT CHR$129;"O"
::GOTO 1880
 1860     IF P%=2 THEN PRINT CHR$135;"O"
;:GOTO 1880
 1870     PRINT CHR$131;"+";
 1880     NEXT
 1890   PRINT TAB(35,18-Y%);CHR$132,Y%
 1900   NEXT
 1910 PRINT TAB(0,18);axis$
 1920 PRINT TAB(0,19);CHR$133;CHR$157;CH
R$131;"My last move was:";
 1930 PRINT CHR$134;FNint_to_char(locati
on%);TAB(30);atari1$
 1940 PRINT TAB(16,22);atari2$;CHR$7
 1950 ENDPROC
 1960 :
 1970 REM*******************************
 1980 :
 1990 DEF FNinput(P%,M%,W%)
 2000 LOCAL S%,A$,I$
 2010 PRINT TAB(39,P%+1);STRING$(119,CHR
$127);
 2020 A$=""
 2030 PRINT TAB(3,P%);mess$(M%);CHR$134;
 2040 PRINT STRING$(W%,CHR$32);STRING$(W
%,CHR$127);
 2050 *FX21,0
 2060 I$=GET$ : IF ASC(I$)=13 THEN GOTO
2120
 2070 IF ASC(I$)<>127 GOTO 2090
 2080 IF S%>0 THEN S%=S%-1:A$=LEFT$(A$,S
%):GOTO 2110 ELSE GOTO 2060
 2090 IF I$>="a" AND I$<="z" THEN I$=CHR
$(ASC(I$)-32)
 2100 IF S%<W% THEN S%=S%+1:A$=A$+I$ ELS
E GOTO 2060
 2110 PRINT I$; : GOTO 2060
 2120 =A$
 2130 :
 2140 REM*******************************
 2150 :
 2160 DEF PROCmessage(P%,M%,A$)
 2170 LOCAL L%
 2180 FOR L%=P% TO P%+1
 2190   PRINT TAB(39,L%);STRING$(39,CHR$
127);
 2200   PRINT CHR$141;CHR$136;CHR$129;me
ss$(M%);A$;
 2210   NEXT
 2220 ENDPROC
 2230 :
 2240 REM*******************************
 2250 :
 2260 DEF FNint_to_char(P%)
 2270 IF P%=0 THEN ="Handicap"
 2280 =CHR$(P% MOD 16 + 64)+STR$(P% DIV
16)+"  "
 2290 :
 2300 REM*******************************
```

Bits On Board

Each position on the board is represented by a single byte in memory. The upper diagram shows the layout of the 255 bytes that represent all the intersections on the 15 by 15 board. Bits within each byte hold information about the state of the corresponding intersection on the board. The two least significant bits determine the colour of the stone present (or indicate whether the intersection is vacant). Bits 2 and 3 will be used by 'game-state' evaluation routines developed later in the project

PLANNING THE ACTION

To conclude our two-part series on program generators, we look at the fundamental operating procedures for two of the most popular packages currently available — Sycero and The Last One. By stressing organised pre-planning, these systems will help you to generate concise and bug-free programs in BASIC.

Any software claiming to enable the complete beginner to generate computer applications must satisfy some stringent parameters:

- It should encourage a 'top-down' approach, with which the user maps out the project in outline and more clearly defines each point as the work continues.
- It should be menu-driven, with multiple-choice prompts that guide the user through the available options.
- It should trap errors as they appear and allow for simple amendments or second thoughts.
- The resulting code should be 'transportable' — able to run on another computer independent of the system software that generated it.
- It should produce ample documentation of the process of program generation, so that a user unfamiliar with the preceding programming can introduce further amendments or enhancements.

However, since the program generator is a tool whose use is not restricted to the uninitiated, it should also be acceptable to the more experienced user. A program aimed at too low a level is likely to infuriate and possibly even confuse the more advanced user, as well as delaying things with unnecessary safeguards and foolproofing. The more experienced user is also likely to have favourite routines — ways of using BASIC, for example — which the generator should be prepared to accept. On all these counts, the most popular program generators, The Last One (TLO) and the newer Sycero, both score high marks.

Informative menus are used throughout in both. Sycero has help screens concerning such features as cursor controls and graphics; text editing commands are available at any time by pressing the Control and H keys. Of the two systems, TLO more closely adheres to the top-down approach. It starts by working out a 'flowchart', which is actually a series of REMarks about what each section of the program is intended to do (Branch on 4-option menu, for example) and can be incorporated in the final BASIC program if required.

Input and display screens are defined during the program coding process and the resulting code will be in ASCII format. The user then has to exit the generator, LOAD the generated program, and re-SAVE it in binary code. In Sycero, file definition and screen design are the first things the user takes care of. This is followed by 'input variable processing' (associating help prompts and error messages with the relevant user input), report definitions and several other options. The various modules will be linked together just prior to coding.

Although this approach is less friendly to the inexperienced user, it has the advantage that it requires the project to be carefully defined before it is coded or it won't run at all. Furthermore, if the generator spots any errors while it is coding, it halts the process and informs you of the error so that you can correct it. The coding process concludes by saving the program, in binary file, which can be run from within the Sycero environment.

Both generators produce fully transportable code. In fact, if the TLO program is short enough, it should be possible to run the ASCII version on an MSX-standard machine (as long as it stays within the micro's memory limitations). However, both programs include very complex and comprehensive error-trapping routines within the programs they generate, making them rather bulky.

CERTIFYING A WORKDISK

As is common with MS-DOS software, both TLO and Sycero must be configured to the hardware in use with Install programs, which can be used with dual-drive floppy or hard disk machines. TLO must also 'certify' a workdisk, a process somewhat like formatting. For this purpose, a workdisk can either be an entire floppy disk or a sub-directory on the hard disk (the latter is certified as part of the installation process). The certification is essential, because if the program cannot detect a workdisk area during flowchart amendment, it will be unable to find the relevant flowchart. This will mean you will have to go back to the beginning or attempt to manipulate the BASIC code. As usual, you can protect against this kind of disaster by frequently making back-up copies of the contents of the disk you are using.

Both TLO and Sycero generate profuse documentation, listing such matters as the screen design characteristics, the variable names used and so on. Sycero adds the date and time on screen as well as on printed copy so that you can distinguish between early versions and later upgrades when checking.

After installation, the TLO user is offered an eight-option Main Dispersal Menu:

Create a program	Enquiry
Modify a program	Certify new disk
Modify a file	Resume coding
File definition	Return to BASIC

After choosing the first option and answering the

DIY Kits
The Last One (TLO) and Sycero both allow the user to generate BASIC programs that will run independently on the host computer. However, precise definitions of the user's requirements involves extensive pre-planning before either package can be used profitably

question: Will your program require files? in the affirmative, you must define the necessary files with their fields and field-types (alpha, numeric or data). After this, the screen will display the Flowchart Creation Menu. On the latest version, 20 options are available, numbered 1 to 12 and 14 to 21 (TLO's author is obviously superstitious):

List flowchart	Clear
Modify flowchart	Set file pointers
Code program	Read a file
Merge flowcharts	Write a file
Abort	Search or sort a file
Keyboard input	Merge
Display data	Record check
Branches	Clear file
Calculations	Database functions
Special functions	Multifunctions

Using these options, you can develop a flowchart, which might result in something like the following, for a names and addresses file:

IAN McKINNELL

1 . .Branch on a 3-option menu
2 . .Set pointer to the end of Address file
3 . .Keyboard input for Address file
4 . .Write data to Address file
5 . .Ask < Have you finished?>. Branch if 'No'
6 . .Direct unconditional branch
7 . .Branch on a 3-option menu
8 . .Set pointer to the start of Address file
9 . .Keyboard search of Address file
10 . .Display data from Address file
11 . .Ask <Another search?>. Branch if 'Yes'
12 . .Direct unconditional branch
13 . .Sort Address file
14 . .Set pointer to the start of Address file
15 . .Read data from Address file
16 . .Display data from Address file
17 . .Direct unconditional branch
18 . .Terminate

A B C D E F G H I J K L

Coding can then commence. Until this point, you have not been able to drop out of the program without losing all the work done so far. As soon as coding has begun, however, it will be possible to quit and use the Resume coding option to re-start. During coding, the destinations of branches are completed — the first menu, for example, will branch to 2 (write data), 7 (read data) or 18 (terminate) — and the screens are designed.

It is possible to SAVE screens, which is advisable, since they can be modified for later use elsewhere in the program or in other programs. However, there is nothing in the software or the manual that gives this feature the emphasis it requires. In Sycero, screen SAVEing is automatic.

SYCERO'S OPENING MENU

Sycero's opening menu offers 13 options that are used in more or less the order they appear:

System configuration
Initialisation
System file-field definition
Screen definition
Screen processing
Report definition
Report processing
Program definition
Generate a program
Create a 'live' data file
Run a generated program
Utilities
End session

By the time you have reached the program generation option much of the work will have already been done, and the fairly simple process that follows will require little intervention, unless the software detects an obvious error. Of course, it cannot deduce your intentions, so it is imperative that you instruct the system in a *precise* manner.

The main difference between the two generators is one of approach. As the Sycero manual says itself, 'The best way to develop a system is to use the top-down approach. Start with the broadest of outlines, and then gradually fill in the details from the top down. Unfortunately, the best way to enter the details into a generator is the other way about, bottom up'.

Because it starts with a flowchart generator, TLO is closer to this classic top-down approach, though paradoxically it may thus encourage the user to start work with inadequate pre-planning. Sycero, on the other hand, is virtually unworkable unless you pre-plan, though it makes it easier to revise things as you go along.

Sycero and TLO both provide the user with a convenient way of generating programs in BASIC to run independently on the host machine. Despite their limitations they can prove useful for the construction of database utilities and simple question and answer calculation programs — although any code generated will be considerably bulkier than an equivalent program written from BASIC.

HOOK LINES

The addition of Interface 1 to the Sinclair Spectrum not only gives access to the Microdrive, a serial interface and local area network (LAN) facility, it also provides us with a few machine code routines that we can make use of in our programs. Here, we take a look at some of these routines.

The Interface 1 contains eight Kbytes of ROM, which occupy the first section of memory. It provides the necessary routines to drive the various additional devices (such as Microdrives), as well as extending the BASIC interpreter to accept such commands as CAT and FORMAT. This ROM is usually called the *Shadow ROM*.

There have been several different versions of this ROM, and there are major differences between the initial implementation (Version 1) and the one currently being produced (Version 2). Version 1 was rather inefficient in its use of Microdrives, and several of the operations that were supposed to be available were not.

Later versions of the ROM are more efficient in handling Microdrives, allowing us to use more of the space on a Microdrive cartridge. Also, the bugs were eliminated, and a couple of new facilities for use with serial printers were incorporated.

These ROM differences are generally 'transparent' to the user, provided that the Interface is accessed using BASIC or the approved machine code programming techniques. However, should you attempt to call Shadow ROM routines directly, you'll probably have problems because different routines are located at different addresses in the various versions of the ROM. Any specific Shadow ROM addresses that we give will be for the Version 1 ROM.

How does the Shadow ROM occupy the first eight Kbytes of memory space when the BASIC ROM is also in this area of the memory map? Well, a technique similar to the paged ROM system of the BBC Micro is employed. Whenever the Z80 attempts to read an instruction from addresses &08 or &1708 with Interface 1 connected, the BASIC ROM is momentarily 'paged out' and the Shadow ROM takes over for future operations. There is a routine in the Shadow ROM to reselect the main BASIC ROM when the time comes.

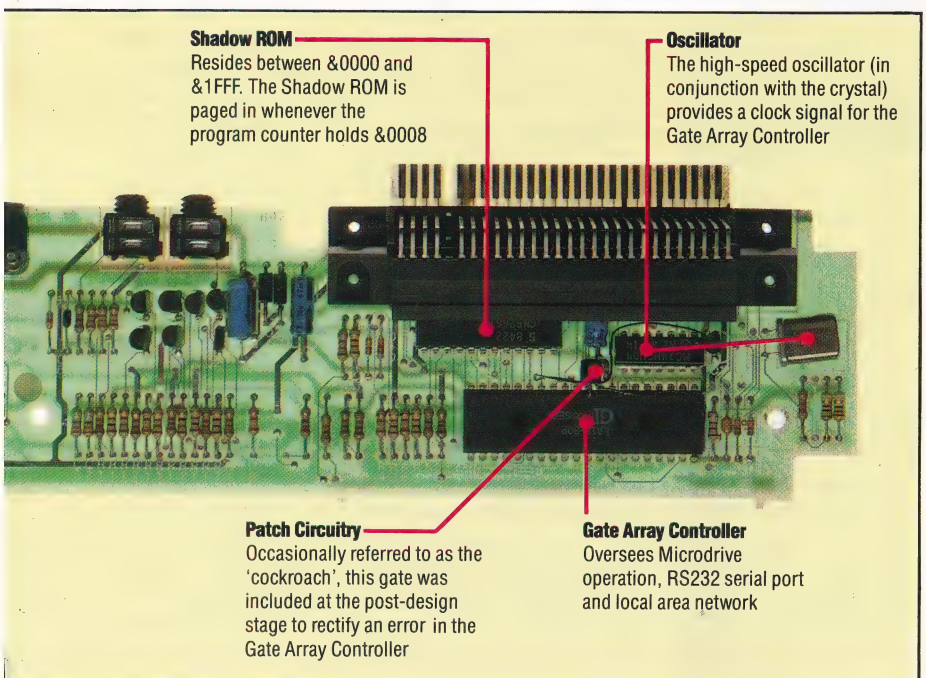
Before any of this can happen, however, Interface 1 must be initialised. This is carried out when:

1. A NEW command is issued with the Interface attached.
2. The Shadow ROM is paged in for the first time.

The practical result of this initialisation process is that a new group of system variables are set up. These occupy RAM from the end of the old system variables to the start of the Channels area — over 50 bytes of space. This, combined with the setting up of the Microdrive map and other information needed by the various devices supported by the Interface 1, is responsible for the fact that with this Interface connected the start of BASIC program text is moved up in memory. This is why it is ill-advised to store machine code in Line 1 REM statements once the Interface is connected (because Line 1 of the program is no longer at a constant position in memory).

As we have already mentioned, Interface 1 also gives us additional BASIC commands. Indeed, we can even add our own, as we'll see in a future instalment of the course. However, at this point we'll just take a simplified look at how the commands such as CAT and FORMAT are implemented. On the unexpanded Spectrum, these would fail the various checks made by the BASIC interpreter, and so in normal circumstances will generate an error. This is done by the use of the RST &08 instruction, thus accessing address &08. When Interface 1 is connected, however, this instruction will page in the Shadow ROM, which will give the offending piece of BASIC a 'second look' to see if it's a command that it can interpret correctly. If it can, it will do. Commands such as the following, which saves the program called Fred

Inner Workings
We show here the principal components of the printed circuit board of the Interface 1. Apart from providing Microdrive, serial interface and LAN facilities, it also makes possible the addition of extra BASIC commands defined by the user. We will discuss user-defined BASIC commands later in the course


Shadow ROM
Resides between &0000 and &1FFF. The Shadow ROM is paged in whenever the program counter holds &0008
Oscillator
The high-speed oscillator (in conjunction with the crystal) provides a clock signal for the Gate Array Controller
Patch Circuitry
Occasionally referred to as the 'cockroach', this gate was included at the post-design stage to rectify an error in the Gate Array Controller
Gate Array Controller
Oversees Microdrive operation, RS232 serial port and local area network

to Microdrive, function in a similar fashion.

```
SAVE *"m";1;"Fred"
```

The * part of the command generates an error condition, thus causing control to pass to address &08 — and hence into the Shadow ROM, where correct interpretation of the statement occurs.

Let's now look at the additional machine code routines that are provided by Interface 1. We'll examine what could be called the 'general purpose' routines — those Shadow ROM routines that are not concerned with the control of the Microdrive, the serial interface or LAN facilities.

HOOK CODES

The first problem is calling these routines; the Shadow ROM is only active after a call to either of the addresses we mentioned earlier. We solve this by making a call to address &08 using the following instructions:

```
RST     &08
DEFB    nn
```

where nn is a value between &1B and &32 (values outside this range generate error messages). These values of nn are called hook codes, and each of these calls a different Shadow ROM routine. The effects of the hook codes have remained unchanged in the different versions of the Shadow ROM. Before we take a look at some of these calls, there are a few points to note.

1. The hook code operations tend to corrupt all the registers, so it is necessary to preserve the registers that you're going to use after the call. In addition, it's a good idea to save the HL register pair, as this is important for a safe return to BASIC.
2. Certain of the hook code routines will turn off the interrupts. Re-enable these if you're not certain.
3. When a hook code call is used, have the value &5C3A in the IY register pair.
4. Before a hook code is used, it's a good idea to ensure that the Interface 1 system variables have been set up. There is, in fact, a hook code to do this job for you, and we'll look at this one first.

- *Hook Code 49 (&31)* is responsible for the creation, or insertion, of the Interface 1 system variables. If you're not certain whether these have been created, then you should issue the following call before attempting to use any of the other hook codes:

```
RST     8
DEFB    49
```

Once a hook code routine has been executed by the Shadow ROM, then control is passed back to the main ROM.

The Shadow ROM also provides us with some routines that are present in a less convenient form in the main BASIC ROM. These deal with input and output to the screen and from the keyboard.

- *Hook Code 27 (&1B)* waits for a character to be typed in at the keyboard, and then returns the character code of the key pressed to the A register. (Note that nothing is printed to the screen.)

- *Hook Code 32 (&20)* examines the keyboard at the instant it is called and indicates — via the status of the C Flag — whether a key is being pressed or not. This routine doesn't wait for a key to be pressed. If a key is being pressed, then the C flag is set to one; if nothing is pressed then it is set to zero.

It's vital that interrupts are enabled before calling hook codes 27 and 32, as the keyboard scanning on the Spectrum is interrupt-driven.

- *Hook Code 28 (&1C)* is responsible for printing the character (indicated by the ASCII code held in the A register on entry to this routine) to stream 2, which is usually the upper portion of the television screen (see page 1332). The use of this routine, and hook code 27, is shown in the following piece of code. If you enter this program, note that it is a permanent loop, and so you'll have to turn the machine off to get out of it.

```
      ;call routine in Shadow Rom - use with care!
210000      ld   hl,ADDRESS          ;insert address of routine
22ED5C      ld   (23789),hl          ;put address in system variable
CF          rst  #08
32          defb #32                 ;do it
```

- *Hook Code 31 (&1F)* is similar to hook code 28, but writes the character (with ASCII code in the A register) to stream 3 — usually the ZX printer.

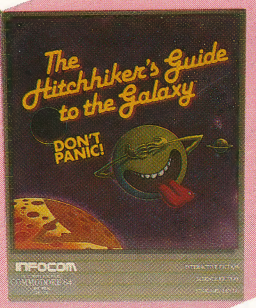
- *Hook Code 50 (&32)* is the final general purpose hook code that we'll look at here. This is rather mysteriously labelled Sinclair Research Use Only in the specifications. This allows us to call a Shadow ROM routine at a particular address from the main ROM, rather than by using hook codes. Because of the differences between versions of the shadow ROM any programs that use this call to access is placed routines will crash if they are used on another version. However, we include details here for the sake of completeness.

The address of the Shadow ROM routine we want to access is placed in two addresses in the system variables produced by the Interface 1. The address of the variable concerned is locations 23789 and 23790. The hook code call is then made. The following section of code will do this:

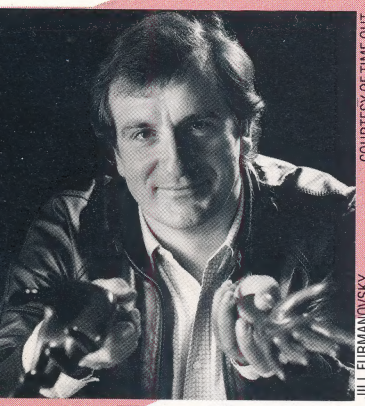
```
      ;wait for char, print to stream 2
      ;Hook Codes #1B and #1C
CF    start:  rst  #08              ;set up Interface 1...
31            defb #31              ;...system variables
FB    loop:   ei                    ;ei - just in case
CF            rst  #08              ;wait for a char...
1B            defb #1B              ;...from the keyboard
CF            rst  #08              ;char now in A reg...
1C            defb #1C              ;...so print it
18F9          jr   loop             ;round again ad infinitum
```

This isn't a terribly useful call unless you know what the various Shadow ROM routines do and which versions of the Shadow ROM you're working with. However, we will examine this call in a little more detail when we come to discuss how we can use Interface 1 to add new instructions to BASIC. In the next instalment, however, we'll look at those hook codes that are specific to Microdrive operations.

INTERSTELLAR INTERACTION



Classic Adventure
The Hitch Hiker's Guide to the Galaxy is the latest in a long series of enormously successful Infocom adventures that began with the Zork Trilogy, three games involving treasure, magic and mystery in the Zork Empire



JILL FURMANOVSKY COURTESY OF TIME OUT

Fertile Imagination
Douglas Adams, author of The Hitch Hiker's Guide to the Galaxy. Originally written for radio, the series has since been printed by Pan Books, with worldwide sales running well into seven figures

Because British programmers have had to produce adventure games predominantly on cassette, their American counterparts now lead the way with much more comprehensive disk-based software. We look at some of the differences in the two approaches and examine a new import from Infocom that should prove popular.

Hitch Hiker's Guide To The Galaxy: For the Apple, Apricot, IBM and Atari range of computers. A Commodore 64 version will shortly be available
Price: From £29.95 (Atari) to £34.10 (IBM)
Distributors: Softsel, Softsel House, Sion Gate Way, Great West Road, Brentford, Middlesex, TW18 9DD
Joystick: No
Format: Disk

Adventure games require large amounts of data storage and are ideally suited to disk-based systems, in which new location descriptions, messages and command routines can be loaded in from disk when needed. This situation has always been something of a problem for British adventure programmers, who have had to write games for a market that has traditionally bypassed disk storage in favour of the less capable cassette.

For this reason, almost all British adventure software is RAM-based and, consequently, rather limited, although programmers have become adept at exploiting the limited space available. In particular, the development of 'interactive characters' (which require large amounts of data) has been held up, as has the introduction of games with large vocabularies.

In the United States, however, the situation is very different. All popular home micros have disk systems readily available, and the generally higher levels of disposable income have meant that such systems are almost always purchased by the first-time user. The market has therefore been ideal for the development of complex adventure software or, as some prefer to call it, 'interactive fiction'. The American software house Infocom has been a leader in providing for this market.

The latest product from the Infocom stable is typical of the high standards now expected from the company. A collaborative venture between Infocom programmers and Douglas Adams, The Hitch Hiker's Guide to the Galaxy is now available in this country for a range of machines, including the Apple IIe and the Atari range of computers (a version for the Commodore 64 will shortly be available). It requires a single disk drive and a lot of patience.

The game is closely based on the series originally written for radio by Douglas Adams, and features such characters as Ford Prefect, Zaphod Beeblebrox, numerous aliens, and, of course, Arthur Dent, the anti-hero of the story who finds himself one day plucked from suburban existence and spirited away into deep space inside the hold of a Vogon Starcruiser.

The first and most obvious strength of the game is the parser — that part of the program that accepts and interprets the user's input. Infocom parsers can differentiate between adjectives, adverbs and prepostions as well as the more traditional verbs and nouns to which most British adventure games are limited. Furthermore, the vocabulary is extremely large — between one and two thousand words — and input can be in a number of different formats. For example, you can enter a direct command (such as Drink the beer), a multiple command (Take the babel fish then put it in the pocket), or even a direct question (Where am I?).

Characters can be addressed by simply saying their names, as in Ford, where are we? or Marvin, go away. Even if the computer doesn't understand exactly what is intended of it, the program will usually come up with an acceptable response – a great improvement on the traditionally British You can't do that or just I don't understand.

Other programming techniques incorporated into this game include the provision of 'containers' (objects that can hold other objects), a facility often missing from British games. Other provisions are 'global objects' (such as 'floor', 'wall' and so on), which may be present in many different locations, and 'vehicles', containers that can hold and carry the player from one location to another. It is interesting to note, however, that most of these facilities, together with the large vocabulary and efficient parser, result from the increased data storage provided by disks, rather than clever programming.

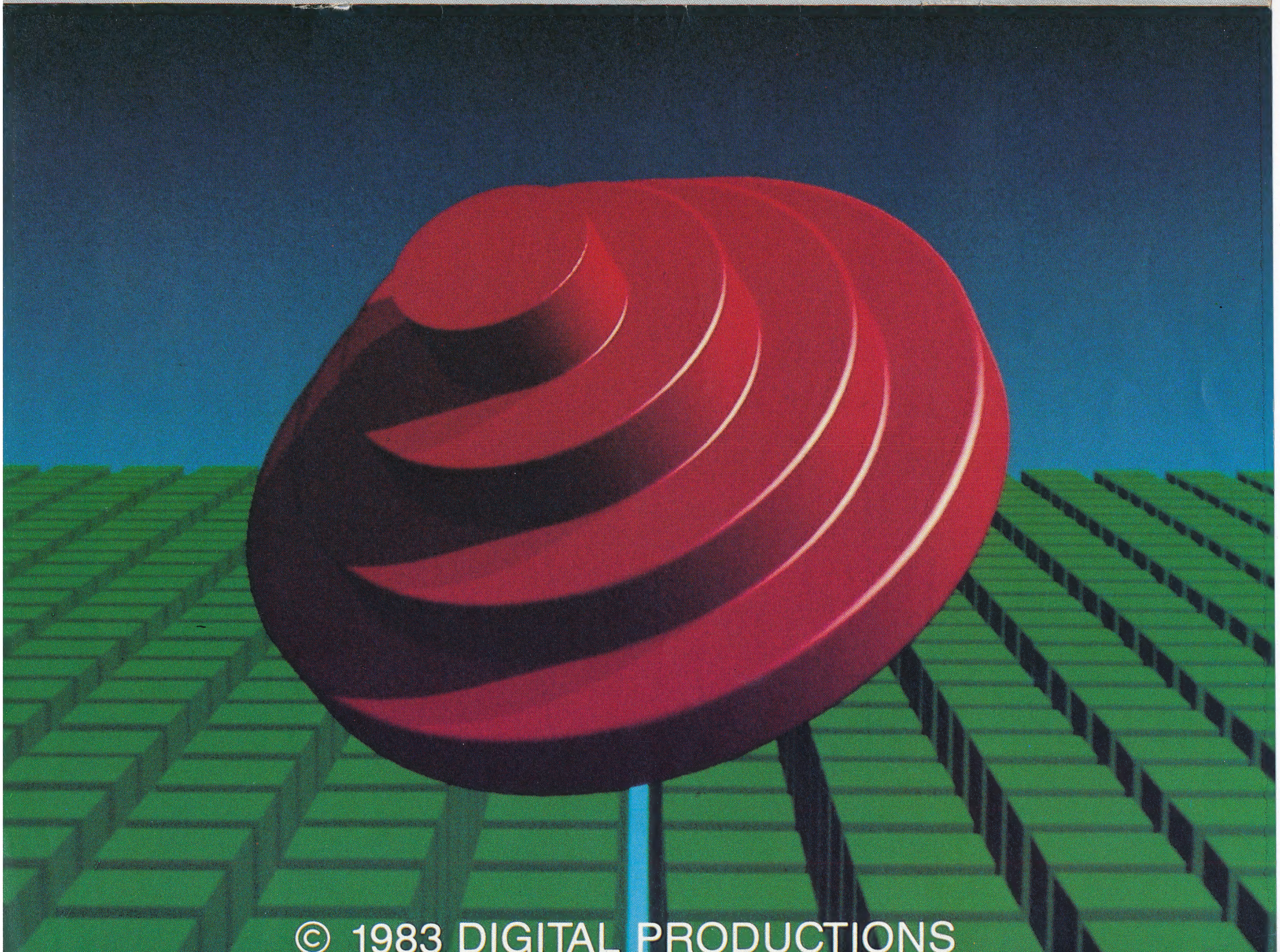
Disassembling an Infocom game reveals a very complex plot and script design, but little in the way of what we have come to call artificial intelligence. There is a strong view amongst some British adventure programmers that the luxury of disk-based programming has made their American colleagues somewhat lazy, and that when the British market abandons cassettes, the compression techniques developed for RAM-based games will result in superior programs.

The Hitch Hiker's Guide to the Galaxy will undoubtedly provide many hours of playing time, and is as full of surprises and humour as the original radio series.

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of the Commodore 64's memory map

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
BAUDOF	0299–029A	665–666	RS-232 Baud Rate: Full Bit Time (μs)
RIDBE	029B	667	RS-232 Index to End of Input Buffer
RIDBS	029C	668	RS-232 Start of Input Buffer (Page)
RODBS	029D	669	RS-232 Start of Output Buffer (Page)
RODBE	029E	670	RS-232 Index to End of Output Buffer
IRQTMP	029F–02A0	671–672	Holds IRQ Vector During Tape I/O
ENABL	02A1	673	RS-232 Enables
	02A2	674	TOD Sense During Cassette I/O
	02A3	675	Temp Storage For Cassette Read
	02A4	676	Temp D1IRQ Indicator For Cassette Read
	02A5	677	Temp For Line Index
	02A6	678	PAL/NTSC Flag, 0= NTSC, 1= PAL
	02A7–02FF	679–767	Unused
IERROR	0300–0301	768–769	Vector: Print BASIC Error Message
IMAIN	0302–0303	770–771	Vector: BASIC Warm Start
ICRNCH	0304–0305	772–773	Vector: Tokenize BASIC Text
IQPLOP	0306–0307	774–775	Vector: BASIC Text LIST
IGONE	0308–0309	776–777	Vector: BASIC Char. Dispatch
IEVAL	030A–030B	778–779	Vector: BASIC Token Evaluation
SAREG	030C	780	Storage for 6502 .A Register
SXREG	030D	781	Storage for 6502 .X Register


© 1983 DIGITAL PRODUCTIONS